UNIVERSITY OF TECHNOLOGY, SYDNEY

SCHOOL OF ELECTRICAL ENGINEERING

THESIS 2

Report Submitted in Partial Fulfilment
of the Requirements for the Degree of
Bachelor of Engineering (UTS) in
Computer Systems Engineering

Congestion Control in Wide-Area TCP Networks using BONeS

Matthew Gream (90061060) - <M.Gream@uts.edu.au>
Academic Supervisor: Tamara Ginige
Spring, 1995

PREFACE

This thesis has been carried out in Partial Fulfilment of the Requirements for the Degree of Bachelor of Engineering (UTS) in Computer Systems Engineering. The thesis is project based and carried out as the final stage in the degree course. In essence, it acts as the culmination of skill and knowledge obtained during the degree.

The thesis is partitioned into two subjects, each being a semester in length. The first subject, Thesis 1, consisted of an investigation and analysis resulting in a requirements definition for the work to be carried out in the second subject, Thesis 2. The report generated as part of Thesis 1, detailing the investigation, project plan and initial design, is appended to this report for reference purposes. During Thesis 2, the work towards the objectives identified in Thesis 1 was carried out, and this report specifically addresses this work, and goals.

This work is concerned with the examination of Wide-Area TCP congestion control using the BONeS modelling and simulation package. As such, it required the construction of an environment within which the models and simulations could be built and executed. This construction consumes a significant part of the work, more so in light of the inability to complete work in the second part. In the second part, issues relating to Wide-Area TCP congestion control were examined, with models and simulations devised to pursue these issues. The intended simulations could not be run and analysed, so the work consists of specifications for the models and simulations, and the related discussion and expected results.

The inability to complete all objectives was due to the fact that this particular thesis was subject to a set of extraordinary events. A situation developed in which BONeS, around which this thesis is based and heavily dependant upon, became unavailable for use. Therefore, this prevented the core objectives of the work being attained, along with delaying final completion and causing personal distress. These events are documented as part of this report.

As a result, there are two significant achievements in this work. The first achievement is a well designed, fully documented, BONeS environment, specially constructed to support simulations utilising the Transmission Control Protocol, but with sufficient modularity to be used elsewhere. The second achievement is the identification and specification of Wide-Area TCP congestion control simulation scenarios, complete with expected results.

ACKNOWLEDGMENTS

I was assisted by a number of people during my execution of this work, and to them I must extend my personal gratitude.

First and foremost, I must thank my thesis supervisor Tamara Ginige. She was prompt, efficient and complete in answering all queries and issues that I presented to her, especially during the unfortunate circumstances that disrupted the work. As such, there were one or two occasions where I was considerably distressed, and I thank her for the continual patience and support she displayed.

Geoff Ingram ("Jellybean") deserves special mention for his hard work in attempting to resolve the problems that persisted with BONeS, and for having to put up with the absurd Comdisco policy that prevented licence renewals. His quick replies and continual status reports were refreshing.

I must thank other UTS staff, notably Warwick Symons, for copies of BONeS manuals, and Peter Yardley, for laboratory facilities on Level 22, for their assistance.

Mention must be made of my employer, Jtec (R&D) Pty, Ltd for providing the type of flexible working hours and environment supportive of part time students.

And finally, a thank you to all the other people that are not mentioned here; this includes personal friends and associates, some of which had to put up with my dedication to work, occasional distress and partial reclusiveness.

ABSTRACT

Wide-Area Networks (WANs) are changing in character, but the Transmission Control Protocol's (TCP) congestion control mechanisms have remained fundamentally the same. This new character is suspected to impact upon the performance of TCP congestion control. This work uses the Block Oriented Network Simulator (BONeS) package to construct models representative of such new environments, and perform simulations involving scenarios of interest.

The work consists of two parts, the completion of which was upset due to unforseen problems with the BONeS package.

In the first part of this thesis, a number of generic and extensible components are constructed for use in modelling and simulating Wide-Area TCP Networks using the BONeS package. These components include Hosts, Traffic Generators and LANs that further decompose into OSI based modules, including Datalink Layers, Network Layers and Transport Layers. An implementation of the Transmission Control Protocol (TCP) as used in BSD 4.4 / Net3 was carried out and forms part of this.

The BONeS environment was constructed, and complete design and implementation details are provided.

The second part of this thesis is concerned with modelling and simulating Wide-Area TCP congestion control scenarios by using these components with the BONeS package. There are five scenarios in total. The first two examine the basic behaviour of the TCP congestion control. The third looks at the effects of multiple paths and dynamic routing in complex WANs, the fourth examines the effects of WAN overloading and TCP congestion control's minimum window size, and the fifth questions the effects of changing WAN traffic characteristics. These last three scenarios are considered potential concerns for contemporary WAN environments.

Models and simulations were developed and specified, including a treatment of our expectations in terms of gathered results. However, no simulations could be executed due to the unforseen circumstances that occurred with the BONeS software. These problems are documented as part of this work.

From the expectations, we find that there is justifiable concern for the performance of TCP congestion control in these new WAN environments. This calls for more detailed investigation, along with the development and assessment of solutions. Several potential directions are discussed in this respect.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACK	Acknowledgment
AI/MD	Additive-Increase Multiplicative-Decrease
ATM	Asynchronous Transfer Mode
BONeS	Block Oriented Network Simulator
BW*D	Bandwidth-Delay Product
ECN	Explicit Congestion Notification
ES	End System
FTP	File Transfer Protocol
HTTP	HyperText Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol (V5)
IPv6	Internet Protocol (V6)
IS	Intermediate System
LAN	Local Area Network
NNTP	Network News Transfer Protocol
OSI	Open Systems Interconnection
RED	Random Early Detection
RFC	Request For Comments
RTT	Round-Trip Time
SMTP	Simple Mail Transfer Protocol
TCP	Transmission Control Protocol
Telnet	Network Terminal Protocol
WAN	Wide-Area Network
WWW	World Wide Web

BACKGROUND, APPROACH AND SCOPE OF WORK

The work in this Thesis covers several fields. The following sections are intended to provide background information on these fields in order to summarise basic concepts from the field, and to capture the aspects that are relevant to this work. Although provided in summary, references can be pursued for more detailed treatment.

The investigation and research for this work was performed as part of the activity carried out in Thesis 1, the report for which is provided in Appendix 3. Thesis 1 also involved the identification of issues and objectives to be pursued in this work. A summary of Thesis 1 activity is also provided in the following sections.

To describe the events that occurred during Thesis 2, especially those that disrupted the completion of this work, a summary of Thesis 2 activity is also given.

1. Background

1.1. Congestion Avoidance and Control

Congestion control is concerned with the allocation and use of resources in a network (Jain, 1990). Resources are in demand by users of the network, however a resource has a finite capacity. The demand for these resources must be controlled otherwise over-demand occurs, at which point the network becomes congested. The two primary resources in a network are transmission links, with finite bandwidth and propagation delay characteristics, and queues, with limited buffering space.

Congested networks drop packets, and therefore result in increased retransmission levels and lower throughput. The goal of a congestion control mechanism is to attempt to prevent congestion from occurring, but if it does occur then be able to recover from it. There are many desirable properties that a congestion control mechanism should have, such as being fair, responsive and having low overhead (Jain, 1990).

In a strict sense, there are two parts to congestion control: avoidance, and control.

Congestion avoidance is a proactive measure. It attempts to prevent congestion occurring by ensuring that the demand for network resources does not exceed the capacity of the network resources. This may take the form of ensuring that the maximum rate of transmission for packets never exceeds the maximum rate at which the network can transport them. This rate is a function of the network’s bandwidth, propagation delay, and queue characteristics as they are shared between multiple users.

However, congestion can and does occur, so congestion control must instrumented at some point. This is a reactive measure, and it attempts to recover from congestion. For example, this may take the form of instructing offending transmitters to reduce the rate at which they supply packets to the network. In the case of Frame Relay networks, Backward Explicit Congestion Notification (BECN) information is used to inform a Frame Relay Access Device that it should reduce its transmission rate (Stallings, 1993).

Throughout this work, the term “congestion control” refers both to avoidance and control except where otherwise stated.

Congestion control is implemented within the network, and can be placed in the end systems, intermediate systems or distributed between both. In the case of TCP congestion control, the end systems, and particularly the transmitters, carry out the implementation. The intermediate systems and end system receivers play a passive role. However, in the Internet Protocol (IP) (Postel, 1981a) suite, the Internet Control Message Protocol (ICMP) (Postel, 1981b) defines a “Source Quench” indication that is used, but is now considered inappropriate to do so, to indicate that a particular transmitter should slow down. This is a case of an intermediate point participating in congestion control.

A solution acceptable for one environment is not necessarily acceptable for another environment, or for that original environment after evolution has taken place. The topic has been given considerable coverage, more recent in the context of Asynchronous Transfer Mode (ATM) Networks (Stallings, 1993).

1.2. Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) (Postel, 1981c) is a full duplex, connection oriented protocol that provides a reliable delivery service for unstructured bytes across an unreliable medium. Typically, this underlying medium is a connectionless network layer, which in practice is usually the Internet Protocol (Postel, 1981a). Both protocols were designed to work together, however TCP makes little assumptions about its delivery medium, and can work with a variety of media.

The TCP is a sliding window protocol. Each transmitted byte is associated with a sequence number, and receivers can only accept bytes with sequence numbers that lie within the range of a current receive window. Upon reception of bytes with associated sequence numbers, the receiver generates acknowledgements for the transmitter. When the transmitter receives acknowledgements and “window advertisements”, it can advance and expand the transmit window, thereby allowing it to send more bytes. The window ensures that bytes with old or duplicate sequence numbers are detected and rejected.

The transmitter detects that bytes have been lost in the network, due to errors or congestion, by sensing a time out on the reception of acknowledgements for that data (a retransmission timer). More recent implementations of TCP have a fast retransmit procedure that detects three consecutive duplicate acknowledgements, assumes that loss has occurred, and pre-empts the retransmission timer. Either way, the transmitter carries out retransmission by resending all outstanding data from the current window. This is referred to as a Go-Back-N error recovery strategy (Stallings, 1993).

The receiver is capable of receiving segments out of order, in which case it enqueues and reassembles them upon the arrival of subsequent segments that fill in the gaps.

In addition to this data transfer behaviour; the protocol includes procedures for connection establishment and termination, using a handshaking protocol.

1.3. Congestion Avoidance and Control in TCP

The original specification for the Transmission Control Protocol (TCP) did not include congestion control measures; as the problem was not recognised at the time. It was not until Van Jacobson's seminal work (Jacobson, 1988), with the exception of (Nagle, 1984) and (Nagle, 1987) that serious focus was given to the issue. His proposed congestion avoidance and control measures form the basis of the current TCP congestion control, and is specified as a mandatory requirement for Internet Hosts (Braden, 1988).

Since that original work, considerable attention has been given to the topic.

Analysis of existing TCP congestion control algorithms	(Brakmo & Peterson, 1995), (Danzig et al, 1995), (Floyd, 1995), (Jacobson, 1990), (Wang, 1992), (Zhang et al, 1991)
New/modified TCP congestion control algorithms.	(Brakmo et al, 1994), (Wang, ??), (Wang & Crowcroft, 1991), (Wang & Crowcroft, 1992)
Focus on TCP congestion control and network participation.	(Floyd, 1991a), (Floyd, 1991b), (Floyd, 1994), (Floyd & Jacobson, 1992), (Floyd & Jacobson, 1993)

Sally Floyd is notable for her extensive work encompassing the end system behaviour of TCP, along with the participation of the network. In general, the majority of the work is focused upon the specific congestion avoidance and control algorithm used by TCP, either to look at the existing mechanism or to suggest modifications and alternatives. Although window based, attention has been given to rate based control (Huynh et al, 1991).

This work is based upon the BSD 4.4 / Net3 TCP implementation (Berkeley Software Distribution, 1994). It uses the original Van Jacobson algorithm (Jacobson, 1988) with a few modifications (such as, "fast retransmit" and "fast recovery") (Jacobson, 1990). A brief summary of TCP congestion control is provided in the following paragraphs, but (Stevens & Wright, 1994) should be consulted for a more detailed examination.

> TCP congestion control is based upon closed loop feedback. It limits the amount of data that can be sent into the network through a *congestion window*, and increases the congestion window as the conversation progresses, therefore placing ever more data into the network. At some point, loss occurs due to congestion, and the *congestion window* is reduced in value.
>
> More specifically, the TCP sender maintains a state variable called the *congestion window*. It always transmits the minimum of the receiver's advertised window, and the *congestion window*. Initially, the *congestion window* is set to one segment, and it doubles each round trip time (i.e. exponential increase) as acknowledgements are received back through the network. The exponential increase occurs until the *congestion window* reaches the value of another state variable: the *slow start threshold*. Initially, the *slow*

> *start threshold* is set to the maximum possible window. This first phase is known as the *slow start* phase (Jacobson, 1988)
>
> When the *congestion window* is greater than the *slow start threshold*, it increases by one segment each round trip time (i.e. linear increase) in an attempt to slowly probe for the networks actual operating point. This operating point is where maximum utilisation of the network's resource is being made, and therefore where the onset of congestion occurs. This second phase is known as the *congestion avoidance* phase (Jacobson, 1988).
>
> Congestion is detected by the loss of packets in the network. The receiver picks this up by a retransmission timeout, or through the reception of three consecutive duplicate acknowledgements (the "fast retransmit" algorithm (Jacobson, 1990)), which suggests that a segment has been lost (but it could have been out of order). When this occurs, the *slow start threshold* is reduced to one half the current *congestion window*, as the *congestion window* is assumed to have been at valid operational point before its last doubling. The *congestion window* is eventually set to the same value as the *slow start threshold*, but may be temporarily increased to instrument a "fast recovery" algorithm (Stevens & Wright, 1994) (which attempts to keep data in transit after loss has occurred).
>
> If the sender has been idle for more than one round trip time, the *congestion window* is set to a size of one segment; since without recent feedback, the state of the network is not presumed known.
>
> During *slow start* the sender is rapidly increasing its *congestion window* in an attempt to quickly reach a presumed stable operating point: the *slow start threshold*. Having reached the *slow start threshold*, it must slowly probe the network until maximum utilisation occurs; i.e. the point at which congestion occurs. Because of this cyclic nature, the transmitter oscillates around the operating point. If the sender increased the *congestion window* linearly from one, convergence would take too long; if it increased the *congestion window* exponentially all the time, losses would be high.

This algorithm is part of a class referred to as "Additive Increase, Multiplicative Decrease" algorithms. These perform additive increases when gaining resources (increasing the congestion window linearly), by multiplicative decreases when releasing resources (decreasing the congestion window by factor of two reduction). Hence, high bandwidth users lose proportionally on the descent, but gain equally on the ascent, leading to a fair distribution of bandwidth (Chiu & Jain, 1989).

Implicit in the above discussion is the centrality of the round-trip times to the mechanisms. Feedback from the network is on a per round-trip time basis, therefore window modifications occur with this period. Retransmission timeouts also rely upon estimated round-trip times.

1.4. Modelling and Simulations

For many reasons, it is often not possible to examine an issue as it occurs in its real problem space. In the case of network situations, it may be difficult to obtain actual measurements, and when these measurements are obtained, the key material may be obscured. In these cases, models and simulations are used to examine issues.

The concern with modelling a problem is to ensure that the model is accurate, especially when the model is going to be used as a basis from which analysis and conclusions are made. The model is only intended to capture the critical defining aspects of the problem. For example, in the case of a network situation, the concept of a datagram may be modelled. However, the actual data in the datagram is not required, only a notion of its length.

A simulation takes the static model and provides it with behaviour; it therefore turns a static model into a dynamic model. The same validity concerns exist with simulations as they do with models: the parameters used in the simulation most are valid and representative.

In the context of Congestion Avoidance and Control, models and simulations have been a primary tool for research, evidenced in (Floyd, 1991a) as an example. Concerns about the validity of such models and simulations have been raised in (Danzig, 1995).

1.5. Block Oriented Network Simulator (BONeS)

The Block Oriented Network Simulator (BONeS) is a software package that provides an integrated environment for the modelling and simulation of networks[1] (Comdisco Systems Inc, 1993) (Shanmugan et al, 1988). It has four main aspects:

- **Data Structures** -- Data Structures are used to hold information used within the simulation. These include primitive types, such as REALs and INTEGERs, along with constructed types, such as SETs and COMPOSITEs. BONeS organises Data Structures in a hierarchy, where each Data Structure is a subtype of its parent, and in the case of COMPOSITE types, also inherits fields from its parent. Users can add and modify Data Structures using a Data Structure Editor.

- **Blocks** -- Blocks are used to process Data Structures. A Block has a number of input and output ports through which Data Structures can flow. Additional Data Structures can be stored and accessed through parameters, similar to the concept of function arguments. Users can add or modify Blocks using a Block Diagram Editor. Blocks can be connected together, to pass Data Structures between each other. BONeS provides a set of basic Primitive Blocks. When necessary, Primitive Blocks can be constructed using the ‘C’ language.

- **Simulations** -- Simulations are used to observe and capture information from the dynamic operation of Blocks. A Simulation Module is constructed and executed with set parameters, and through the use of “probes”, can be requested to capture operating information. Simulations are configured (by the insertion of probes and the specification of parameters) and executed using a Simulation Manager.

- **Post Processing** -- Post Processing takes the information collected from a Simulation and allows for it to be filtered, processing and converted into graphs. These graphs are then used for analysis purposes.

[1] BONeS is even more general than this, it can be applied to many different problem domains, which has included spread spectrum analysis.

To carry out a simulation in BONeS, the user first defines the Data Structures to be used, possibly including “packets” or other suitable representations from the problem domain. Blocks are then constructed to operate upon the Data Structures and model the problem, until a single Block encapsulates a model of the entire problem. This is then defined to be the Simulation Module, which is configured with Probes. The Probes capture Data Structures from an active simulation and write them to a file. The simulation is executed, after which post processing is used to generate the necessary graphs. Conclusions are then drawn based upon analysis of the results.

BONeS has a number of benefits. It is easy and fast to use, and allows for all aspects of a simulation to be constructed in the one package. Iterations and variations of simulations can be executed effortlessly. For advanced use, BONeS provides an interface to the ‘C’ language. This allows for complex or specialised Blocks to be implemented in ‘C’, which is generally much faster execution wise and potentially much faster development wise, with a minor flexibility trade-off.

BONeS has been used before, both at the University of Technology, Sydney (UTS) and in other research (Shanmugan, 1988).

1.6. Wide-Area Networks

Wide-Area Networks (WANs) consist of Local-Area Networks (LANs) connected across large distances. Typically, the LANs contain network nodes that are interconnected at a relatively high bandwidths; using media such as Ethernet/IEEE802.3 (10Mbps), Token Ring/IEEE 802.2 (4-20Mbps) and FDDI (100Mbps). The WAN connections are often at much lower bandwidths, typically involving media such as DDS (48Kbps), ISDN (64Kbps), T1 (1.44Mbps) and E1 (2.048Mbps).

The Internet is perhaps the best example that can be used to illustrate WANs and LANs. An example of such is the previous architecture at the University of Technology, Sydney (UTS). It involved internal 10Mbps Ethernet LANs, internally connected to each other at 10Mbps. A single external (WAN) connection was available through a 126Kbps ISDN line (two 64Kbps B Channels, aggregated and losing 2Kbps in overhead). This external connection was regularly operating at maximum capacity, indicating that the addition of any new conversations would have experienced congestion.

This is typical: the WAN is generally the bottleneck, and therefore where congestion avoidance and control is primarily targeted (Jain, 1990). Furthermore, traffic profiles on WANs are different from those on LANs (Cáceres et al, 1991). A LAN may be characterised as having many short, bursty connections -- due to the nature of interaction between client and server machines: file/image retrieval, electronic mail, terminal sessions. A WAN, on the other hand, generally has sustained transactions (e.g. file transfer) that last for longer periods of time (Paxson, 1993a).

The recent Internet, however, has seen a gradual change in traffic profiles due to the increasing use of the World Wide Web (WWW). Its session protocol uses the TCP for short requests and responses. Congestion control measures rely on network feedback information, of which there is considerably less with such conversations.

Characteristics of WAN traffic have received notable attention, of which significant proportions have been carried out by Vern Paxson[2].

2. Investigation Concerns and Objectives

After examination of the various fields relevant to the topic of this work, a few issues of concern become apparent. Primarily, issues where chosen that are related to the basic fundamentals of TCP congestion control as they are being challenged by the evolving nature of Wide-Area Networks (WANs).

The following three issues were selected for attention, as they are considered contemporary and relevant.

- WANs are increasing in size and complexity. TCP conversations may now expect to have subsequent segments traverse different paths, where previously most or all segments would traverse a single path. This means that conversation round trip times can vary significantly, and out of order delivery becomes common. The performance of TCP congestion control may be adversely affected by changes in these two factors.
- WANs are increasing in utilisation. A WAN connection may now expect to carry hundreds of simultaneous TCP conversations. With more conversations, each conversation receives a smaller share of the available space in the connection. The TCP congestion control has a minimum rate at which it sends segments into the network. If the conversation is allocated a share smaller than this rate, high levels of retransmissions may occur. This challenges the assumption that TCP congestion control makes about the minimum capability of the network.
- WANs are experiencing a change in traffic profiles. WAN connections are now subject to may short and bursty transaction conversations due to the rising use of the World Wide Web's transaction oriented session protocol. These short and bursty conversations do not exit for a time long enough to gain information on, and therefore co-operative with, network conditions.

These are the three Wide-Area Network TCP congestion control issues under investigation. However these are not the only objectives for this work. In addition, it is an objective that the BONeS modules constructed to service these investigations are presentable and re-usable in order to gain additional value from the work.

In summary, the following are the objectives for this thesis:

- Construct a BONeS environment capable of modelling and simulating congestion control issues in Wide-Area TCP Networks. This environment must be presentable and re-usable.
- Investigate several concerns relating to the nature of congestion control issues as they occur in Wide-Area TCP Networks. These concerns are contemporary in nature and have practical relevance.

[2] It is notable that Vern Paxon, Sally Floyd and Van Jacobson are associated with the same research group at Lawrence Berkeley Laboratory.

3. Thesis 1 Activity

Thesis 1 was concerned with the investigation and definition of work required for Thesis 2. The abstract goal was to examine issues of congestion in Wide-Area Networks, in terms of the Transmission Control Protocol. This would be carried out by modelling and simulating with the BONeS package. Additional requirements were that the environment constructed in BONeS should be of presentation quality, suitable for re-use as a whole, or in part. This would add value to the results of Thesis 2.

The investigation covered an examination of the significant fields that intersected with the topic, including, primarily, congestion control itself, in terms of basic philosophy, theory and principles, but importantly as it occurred in the context of the Transmission Control Protocol. Therefore, it required an understanding of the Transmission Control Protocol, and issues particular to Wide-Area Networks. Finally, the generative issues include Modelling, Simulation and the BONeS package itself.

Material examined included books, research papers, conference proceedings, electronic publications, electronic mailing lists, newsgroups, software source code and even consultation with experience practitioners (through email). The Thesis 1 report is provided in Appendix 3 and it details the material examined (although it does lack mention of software source code and consultation, because they occurred between Thesis 1 and Thesis 2). The investigation provided the indication of the avenue that needed to be pursued, and the elements required for that pursuit. This led to the development of a number of specific objectives and, subsequently, the simulations to executed and analysed.

The final step in Thesis 1 was to carry out an initial top-level design for all elements of the work, specifically for the purpose of being able to gauge the time and effort required for the work in Thesis 2. The result was a project plan, and a complete picture of all aspects of work required, both primarily and ancillary.

After the submission of the Thesis 1 report, the BONeS package was used to implement and simulate a portion of the design -- the Link -- as an effort in familiarisation with the package. This signified the completion of all work in Thesis 1.

4. Thesis 2 Activity (including problems with BONeS Software)

Thesis 2 was subject to extraordinary events. These events prevented the achievement of the central objectives of this work, and at the same time caused personal disruption and distress. These events are briefly documented in the following paragraphs.

Thesis 2 started in August 1995. The first one and a half weeks involved a redesign of the BONeS modules, and a re-examination of the simulations that were to be performed. In the time since the completion of Thesis 1, it had become apparent that the work in Thesis 2 would be better served by a redesign: the altered design would reduce the risk of problems, and further value-add to the end results of Thesis 2, in that not only would the central objectives be reached, but the environment constructed to reach the objectives would of such a nature that it could be re-used.

The implementation of the design was carried out over the last two weeks of August 1995, and through September 1995. The implementation proceeded faster than the original plan, which predicted completion in 8 weeks. However, it had slipped by one and a half weeks due to the redesign. The original project plan provided for 2 weeks to cover uncertainty, so this excess time was still within target.

On the 7th of October 1995, the implementation was almost complete -- the TCP ‘C’ implementation was two days from completion, and then a final two days were needed for its integration and for cleaning up various other bits and pieces. However, upon arriving at UTS, it was found that the BONeS software was unavailable (the licence server was not operating). When still unavailable the next day, the following item of mail was sent to Geoff Ingram.

```
From: Matthew Gream <mgream@heckle.ee.uts.EDU.AU>
To: geoffi@ee.uts.edu.au
Subject: BONeS/mozart.
Date: Sun, 08 Oct 1995 09:37:38 +1000

Hi Geoff,

I'm using BONeS on mozart for Thesis 2. As a result of the unfortunate
demise of schutz (as noted in a motd on mozart) it seems that BONeS is
an application that has suffered :

>  mg(mozart).{~} bones
>
>  Could not obtain license for feature "DESIGNER_FRAMEWORK", version 2.0,
>  because cannot connect to license server.
>
>  Exiting ...

Usage of this software is exceedingly critical for me (I'm a part time
student, so my main use is on the weekends), so is it possible for you
to give me an estimated downtime for schutz so that I can attempt to
try and schedule around the situation ? Otherwise, is it possible to
work around the problem ?

Much appreciated,

Matthew.
```

To fill in the time, work on the report was started, and manual verification of the design, the implementation and the simulations was carried out in an attempt to ensure that any potential problems would be averted. However, this week of unavailability stretched into two weeks, and then three. A decision was made to fill in the time by concentrating on documenting the design work. Then the following item of mail from Geoff provided some hope.

```
From: geoffi@ee.uts.edu.au (Geoff Ingram)
Subject: BONeS
To: tamara (Tamara Ginige), tb (Teresa Buczkowska)
Date: Mon, 30 Oct 95 11:00:05 EST

I have just received the license transfer agreement fromn COMDISCO for SPW and
BONeS.  It is now filled out and signed and faxed back to the US.  I'm
uncertain as to how long it will take to receive the passcodes.  I guess a few
days.
```

However, these few days extended further, and in the second week of November 1995, after 5 weeks of BONeS unavailability, it became apparent that it would not be possible for the simulations to be carried out, there was just not enough time left -- this was then tempered by the following item of mail from Geoff providing more uncertainty about the resolution of the problem.

```
From: geoffi@ee.uts.edu.au (Geoff Ingram)
Subject: BONeS/SPW
To: tamara@ee.uts.edu.au (Tamara Ginige), tb@ee.uts.edu.au (Teresa Buczkowska)
Date: Tue, 14 Nov 95 13:28:27 EST
Resent-To: M.Gream@uts.edu.au
Resent-Date: Tue, 14 Nov 1995 13:59:29 +1000

Tamara, Teresa,
        The latest in the saga of BONeS/SPW;

The Australian agent for Comdisco now says that since we have not had software
maintenance for the above products over the last year we are not entitled
to transfer the license to another machine.
I have got very upset with them and have sent an email and fax to the U.S.
as I regard the agents in Australia as a bunch of ignorant #$%^!@#^*
If I receive no satisfaction from Comdisco I will discontinue support of
SPW/BONeS and install OPNET or other similar packages instead.
I have never before encountered such an attitude from a software vendor.
```

In an attempt to find a work around, I located information on software that could fool the licence server into thinking that it was working on the correct machine. I provided this software to Geoff, but his attempts to have it work failed, the licence software was too smart. At this stage, a decision was made to postpone the presentation, in the hope that when the BONeS problems were resolved, the remaining work could be finalised. Over the next couple of weeks, this report was finalised to a point where all information other than the implementation and simulations were documented. The latter two were not completed due to the possibility of the BONeS software becoming available at a later date.

In December 1995, I attempted to locate someone that would assist me by providing Postscript files of my implementation, so that at least it was possible to document the work that I had achieved. I found a very helpful person who offered to do this.

```
Hi Matthew,

You can put your blocks on our ftp site: makalu.theoinf.tu-ilmenau.de
under the directory pub/incoming. We can try to print your blocks in an
ps/eps file format. You then can fetch it back from our ftp-server.

I hope this will solve your problem.

Best regards

Ulrich Freund

Technical University Ilmenau
```

Due to other commitments, and the need for a break, I did not carry out any thesis work for the first two weeks of December 1995. But on the 17th of December 1995, I received the following mail from Geoff indicating that BONeS was available for use for the last two weeks of December 1995.

```
Well, it works.  When I got back from the reef there was a new mother board
waiting for me.  So I swapped the EPROMS and fired it up and ran BONeS and
it's happy.  If you're going to use it, use it quickly because the license
expires on 1st January 1996 and I'm not renewing it.
```

I determined that these two weeks would not be enough time to complete the simulations in their entirety and to extract the necessary diagrams required for the documentation, so a decision was made to finish the implementation and obtain the diagrams over the week preceding Christmas 1995. I finished this task on Christmas

Eve. For the week between Christmas and New Year of 1995, I was already committed to a long-awaited holiday.

No work was performed during January 1996, due to personal commitments -- which partially included moving house, amongst other things. At the end of January 1996, contact with Tamara Ginige was resumed, and I decided to carry out a presentation on the 4th of March 1996, illustrating the work that I had achieved thus far, and emphasising my plans towards achieving the objectives that could not be obtained due to the unavailability of BONeS. The last two weeks of February 1996 were concerned with finishing this report, and preparing for the presentation.

PART 1. DEVELOPMENT OF THE SIMULATION ENVIRONMENT

1. Introduction

The development of the simulation environment was a significant task in this work, involving a process of design, implementation and testing. Issues such as extensibility, re-useability and presentability were purposely considered as core requirements, as a specific goal was to have an environment that could be used in a multitude of ways; not only for this work, but for other interested parties requiring generic network components.

In carrying out the development of this environment, considerable attention was given not just to what tasks were required, but how these tasks were to be carried out. For this reason, discussion is carried out on methodologies, processes and considerations. Given that we are deeply concerned about future use of this environment, it is further clear that sufficient documentation and justification is essential in explaining not just what the environment is, but also how it came into being.

This chapter is segmented into three main sections. The first covers the design of the environment. This addresses the requirements as an overview (the first report constituted the results of the requirements analysis process) and then details design methodology and considerations before advancing onto a top-level architecture, and thence design of modules within the architecture.

The second section presents the implementation of this design. This process was relatively straightforward, however, there are a number of important issues that are discussed. These include certain ways that specific aspects of the design were mapped into the BONeS environment, along with other process considerations. All aspects of the BONeS implementation are presented, including data structures, block diagrams and custom 'C' code where appropriate. Some are relegated to appendices.

The third section addresses testing. Due to the object nature of the designed environment, it is possible to verify modules as stand-alone entities, occasionally supported by previously verified modules. The testing was informal, but never carried out due to the problems that occurred with BONeS, so a brief discussion is given to outline the testing concerns.

2. Design

The design phase is concerned with first resolving an architecture and then the entities within that architecture. The goal is to meet the requirements of being generic and extensible, along with specifically addressing the needs for the simulations that are to be carried out in this work. Significant focus is given to the former, whereas the latter is largely a result of the former.

2.1. Strategies

The design phase was carried out with specific considerations identified before design commenced. Considerations were mostly concerned with a process to be used in the design, and issues of attention during the execution of that process. The first design issue is a high level architecture, requiring the identification of high level components, and an assignment of functionality.

Having resolved a high level architecture--which must address the requirement for re-useability and extensibility--the individual modules within that architecture are considered. Each requires a detailed design, which is carried out by way of the following steps:

1. It is known what functionality must be provided by the module. This functionality is generally related directly to the inputs and outputs of the systems in a well-defined manner.
2. An examination is made with respect to the real world (if the module is not generative) entity that the module provides a model of. This serves as a starting point for understanding the behaviour of the model.
3. An external interface is developed. The architecture already resolved what modules communicate with other modules, and a abstract notion of the messages to be used in such communication, so this process continues the refinement. These issues intend to give a complete coverage of the boundary conditions for the module: which serves as a starting point for the internal design, involving:
    - Establishing the position of the module in relation to other components.
    - Determining exactly what messages and the content of these messages are to be communicated to support the functionality.
    - Determining externally visible parameters that are affected by the associated entities, or are otherwise available to be utilised.
    - Sketching the behaviour and nature of the inputs and outputs to the module, due to the inputs that result in outputs as a function of the parameters and internal state.
    - Determining what ancillary functions must be provided to external entities to aid them in the processing of inputs and outputs.
    - Determining what ancillary functions must be provided by external entities to aid this module in the processing of inputs and outputs.
    - Indicating what state the module will be in when it first starts.

4. The internal design is developed. This consists of an initial partition of the internal module from which the entire internal functionality is resolved.

In constructing the internal design, the chosen methodology to use is that of data driven functional partitioning. This is due to the nature of the environment that is characterised by the stimulus of an input message whose action can be traced through to cause specific functional responses. The use of structured design methods and notations, such as data flow diagrams and process specifications, is appropriate here for this reason. Also, with data flow diagrams, the mapping into BONeS is extremely straight forward, as BONeS is inherently a data flow processor and processing bubbles transform directly to BONeS Modules. The data flow diagrams can, and do, use control flows as BONeS has the concept of a "trigger" to implement a control flow, further assisting the transition.

Contrast this data based processing system to a state based processing system where inputs are first classified according to state, then processed according to their characteristics. Such a system is not easily amendable to BONeS, because it would involve significant data switching and duplication of modules for actions that are present in multiple states.

In adherence to the concept of information hiding, and in some respects the Object Oriented paradigm, all data used in communication is not directly accessed. Modules that "own" the data provide accessors that allow for creation, manipulation, and destruction of data elements. This encapsulation lends itself to several considerable benefits, not the least of which is the ability to hide processing (e.g. insertion of hidden fields and probes) within accessors. This activity causes the creation of a lot of small modules in a bottom up manner, but this is a trivial exercise that provides many benefits.

In general though, the concept of information hiding is a prime concern. But other engineering principles such as coupling, cohesion, consistency, extensibility and understand ability are also required. These are all observed.

The following points summarise various other strategies taken.

- Placing “stubs” for processing is used in situations where processing does not occur, but there is a chance that either it may be added at a later date, or more importantly, a probe may be placed at such point to capture the data that would be processed. Simulations need to use probes at strategic places to collect data.
- Processing is divided in such a manner that coping with new types of processing, or new elements to be processed requires a trivial, or at least mild, modification or addition. This provides a greater allowance for extensibility, re-use and testing -- the latter a prime concern.
- Functionality is abstracted up towards the top of a sub-module for clearer understanding. A user may need to place a probe at some point, so therefore should be able to navigate an internal construction with ease. If the user cannot easily understand the construction, then incorrect probe placement and results may occur.
- Re-useable elements are located and exploited where possible, this is further aided by BONeS ability to defer typing for a module until it is

connected to neighbouring modules. This allows for generic processing blocks to be constructed; similar to parameterised templates in C++, as an analogy. Work is saved, and testing time is reduced.

- The names of processing blocks are intended to be easily understood and correctly representative of the internal behaviour. Again, a user is expected to be able to navigate for the purposes of placing probes.
- Performance concerns are addressed, this may take the form of implementation in the more primitive 'C' language so as to provide an execution speed-up. Simulations are notorious for consuming considerable time and resources, attempts must be made to minimise this.
- Specific capabilities of BONeS are worked towards, but not made as dependencies in the design. This includes deferred typing, data inheritance, type resolution and module re-use. If the capabilities are present in the implementation environment, then maximising the use of them is beneficial.

An alternate design approach may have been to use an Object Oriented methodology. This could have allowed further re-use (e.g. the processing of primitive messages could be abstracted into a class which is inherited and overloaded by each layer) and a smaller design. However, as BONeS itself does not support several fundamental aspects of the OO concept, a mapping would have been required. It is much easier to use an alternative design strategy; one that best supports the problem at hand.

The documentation of the design is further considered to be important due to envisaged re-use. The data flow diagrams, process specifications, abstract data types and the suchlike are all presented with annotations.

2.2. Architecture

The high level design results in an architecture. This architecture identifies significant components and the interfaces between them; and hence the way in which the components interact. The architecture is of critical importance due to the explicit concerns about extensibility.

The requirements for the design are for network components that can be used to carry out various simulation scenarios. These network components must also satisfy several other criteria. They must be extensible and generic, and not unduly complex in the way they communicate with each other and --importantly--, they need to be easily understood if others are to use this environment and they should try to adhere to good engineering concepts, such as modularity. In summary, it should be possible to use (and re-use) them in a wide variety of situations with an ability to extend and augment them in various ways.

When examining communications networks, it becomes apparent that, fundamentally, there are three distinct components, illustrated in Figure 1-2.1.

- **End Systems** -- These originate and generate traffic; they are connected to a Communications Link via one access point only (in general). They exist as stand-alone systems; an example is a UNIX Workstation, or a desktop PC.

- **Intermediate Systems** -- These are connected to multiple Communications Links via multiple access points. They do not generate (significant) traffic, but serve to switch traffic between the Links that they are connected to. These exist as stand-alone systems; an example is a Router, or sometimes a Workstation configured to operate as a Router of sorts.
- **Communications Links** -- These transport bits of information across a geographical distance. They manifest themselves in the real world as Public Network connections (e.g. DDN, ISDN, Frame Relay, ATM) or other connections (e.g. Ethernet, Fiber Optics, Radio).

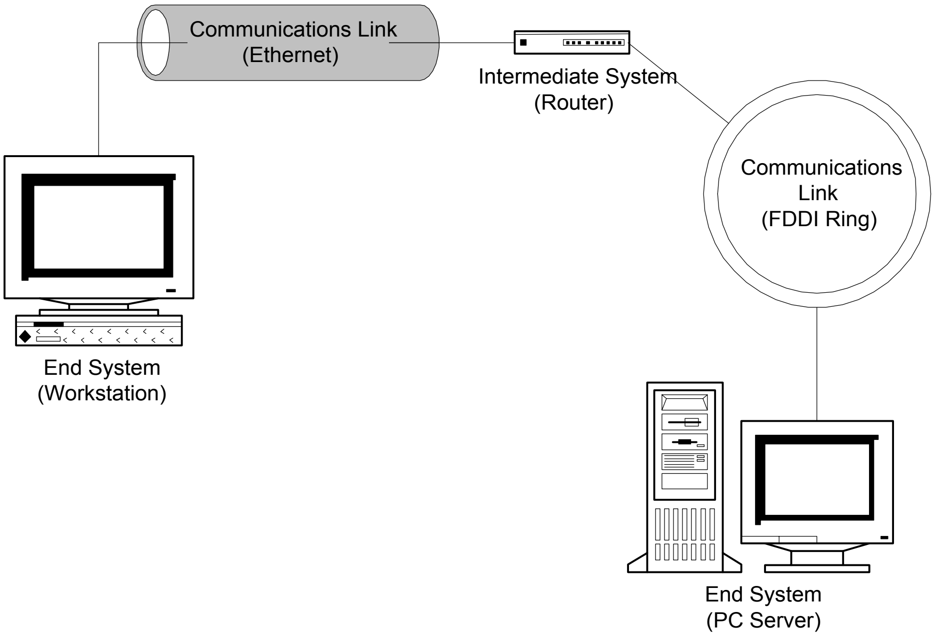


Figure 1-2.1. Fundamental Network Components

This delineation serves as a good starting point since virtually all networks can be described in terms of such components. In addition, the OSI reference model (ISO7498, 1984) uses this terminology as well. The simulation environment is concerned with all three components, plus additional sub-categorisations: e.g. the split of End Systems into "Hosts" and "Traffic Generators"; where the former has a connection oriented capability, and the latter exists to excite the network with traffic on a connectionless basis.

The OSI reference model, in fact, provides an extremely useful framework to use in further partitioning of the architecture. The reference model is concerned with the external view of operation of entities, not so much the internal. The concept used is that of layering, where each layer in the model provides a specified service, and this service is provided though a rigidly defined set of operations. The layers provide specified services at the upper boundaries, but use specified services at their lower boundaries. The importance of this distinction is that it should be, if done correctly, possible to provide a service in an entirely different way by altering or replacing a layer and not have any consequent effect on other dependant layers. In effect, the OSI reference model describes "what" a layer will do, without "how" it will do it.

The seven OSI Layers, shaped though a set of specified principles (International Organisation for Standardisation, 1984), are defined as follows.

1. **Physical** -- Concerned with the transmission of an unstructured bit stream over a physical link; e.g. mechanical, electrical and procedural characteristics. Examples: V.24, X.21bis, I.430.
2. **Datalink** -- Provides for the transfer of data across the physical link; it may use synchronisation, error control and flow control. This is sometimes reliable. Examples: High-level Data-Link Control (HDLC) and Ethernet/IEEE802.3.
3. **Network** -- Provides upper layers with independence from the data-transmission and switching technologies used to connect systems; and is responsible for connections across networks. Examples: Internet Protocol (IP), Connectionless Network Protocol (CLNP), X.25.
4. **Transport** -- Provides reliable, transparent transfer of information between end points; with end-to-end error recovery and flow control. Examples: Transmission Control Protocol (TCP), Transport Protocol 4 (TP4).
5. **Session** -- Provides control structure for communication between applications, e.g. tokens, checkpoints, synchronisation. Examples: File Transfer Protocol (FTP).
6. **Presentation** -- Performs data translation for the purposes of providing a standardised representation across diverse platforms. Examples: Abstract Syntax Notation 1 (ASN.1), External Data Representation (XDR).
7. **Application** -- Provides specific services to the users of the OSI environment. Examples: Remote Procedure Calls (RPC), Electronic Mail (X.400).

Justification for this partitioning can be found in ISO 7498, and for a more complete coverage a detailed reference should be consulted, such as (Stallings, 1993A). A diagrammatic representation of these Layers as they map into the previously identified network components is shown in Figure 1-2.2. The Management entity will be explained subsequently.

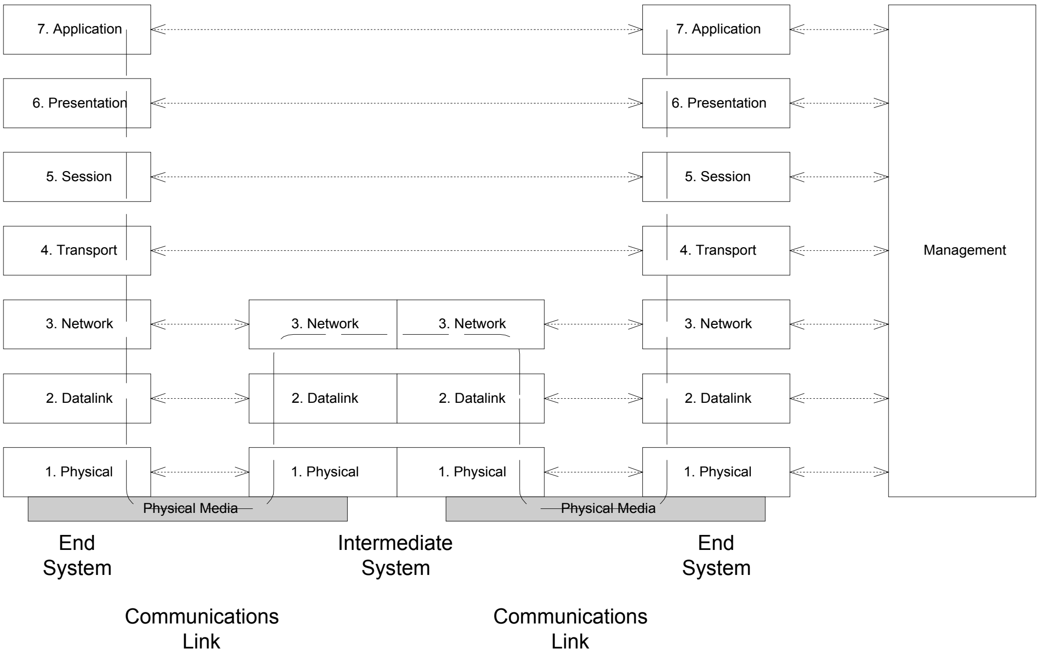


Figure 1-2.2. OSI Model: Layering Concept

In this work, concern is primarily with the bottom 4 layers, as the upper 3 layers exist generally as a specific user service, and are not particularly important from the view of examining the functionality of protocols. Consider the upper 3 layers to merely "add value" to the lower 4 layers, whereas the lower 4 layers implement the critical core functionality.

For communication between layers, a set of conceptual primitives and parameters has been defined. The primitives specify the functions to be performed and the parameters qualify those functions with data and control information. The following primitives are defined, as illustrated in Figure 1-2.3.

1. **Request** -- Issued by a Service User to invoke a Service, and pass parameters to, a Service Provider.
2. **Indication** -- Issued by a Service Provider to indicate that a procedure has been invoked by the peer Service User on an association; or to notify the Service User of an action initiated by the Service Provider.
3. **Response** -- Issued by a Service User to acknowledge or complete a procedure previously invoked by an indication to that Service User (by the Service Provider).
4. **Confirm** -- Issued by a Service Provider to acknowledge or complete a procedure previously invoked by a request to that Service Provider (by the Service User).

Note that a Service Provider consists of a Layer (N) instance, and a Service User consists of a Layer (N + 1) instance. The inter-layer communication (between a peer Layer (N) and Layer (N)) is layer specific.

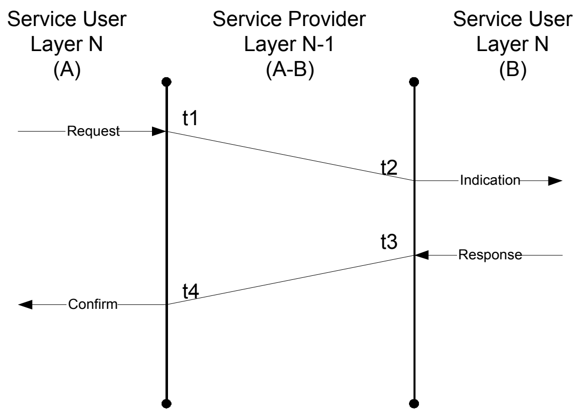


Figure 1-2.3. OSI Model: Message Primitives

In this work, concern is largely with Requests and Indications, simply by virtue of the current functionality--much of it is unconfirmed; the only confirmed behaviour is with the transport protocol which coming from a pre-OSI era does not strictly map into this environment.

By using the OSI Reference Model, by way of the Layering Concept, the following can be satisfied:

- **Genericity** -- The layers can be used in a variety of situations. They are not tied to a specific type of communications architecture. The same layer can be used in different systems, and the Service functions are not oriented towards particular ways that the layer's functionality is provided internally.
- **Extensibility** -- The layers can be internally modified to provide more functionality or different types of functionality. New layers can be developed and can slot into the place of existing layers with no, or at least trivial, modification. The rigidly defined interfaces ensure this.
- **Coupling and Cohesion** -- The inter-layer communication is strictly defined and minimal. It is abstracted to be sufficient for most scenarios. The defined primitives for communications are clear and clean abstractions.
- **Consistency** -- The inter-layer communication has abstract elements that have similar semantic and syntactic content, regardless of layer being communicated with.

What this means is that new layers can be easily developed and inserted, and in general the constructed environment can be expanded and (re-)used. The construction of complex networks is trivial, because the interfaces between components are simple, yet extensible. The direct correspondence between elements in the real world, and elements in this environment, by way of the OSI paradigm, allows for the direct translation of problem domain aspects into this space for solution--extremely critical in a modelling process.

Having decided to base our architecture upon this model, the next issue concerns the way in which it is used in this work. Doing so means considering each of the required components--the *Communications Link*, *Host*, *Traffic Generator* and *Intermediate System*--and how they could be constructed using the OSI model. The method taken was to build these components out of entities that mapped directly to OSI layers where at all possible.

The first component to consider is the *Communications Link*; this link connects *End Systems* and *Intermediate Systems* and generally encompasses the *Physical Layer* and the *Datalink Layer*.

There is no need to model a *Physical Layer* here, so the *Communications Link* is constructed of a *Datalink Layer* that internally models a *Physical Layer* (i.e. transmission delay via. bandwidth and propagation characteristics). Thus, the *Communications Link* is illustrated in Figure 1-2.4.

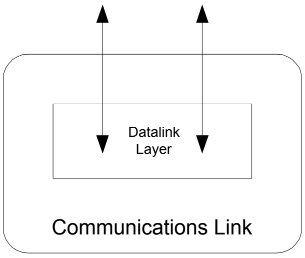


Figure 1-2.4. Communications Link

Secondly, the *Host* component requires a transport service, as this is the prime element in the environment (the work is concerned with transport level congestion control!). The transport service occupies the *Transport Layer*, but also requires a traffic stimulus. The stimulus is generated by an Upper Layer entity referred to as a *Generator*.

In order to abstract the functionality of this Generator away from being a "specific" Transport Layer Generator, and to avoid unnecessarily coupling the *Generator* with the *Transport Layer*, some adaptor is needed. This adaptor is a *Transport-Adaption Layer*. At the lower edge of the *Transport Layer* a *Network Layer* is used to provide access to the network. This *Network Layer* connects to the *Communications Link*, which has been defined to be a *Datalink Layer*. The *Host* is shown in Figure 1-2.5.

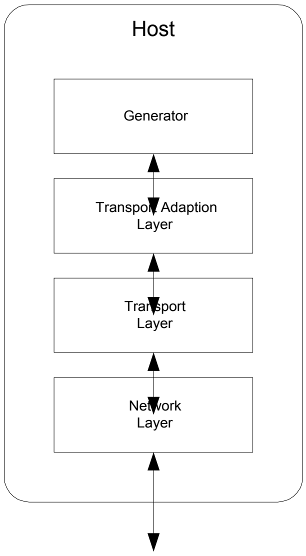


Figure 1-2.5. End System: Host

Thirdly, the *Traffic Generator* is an *End System* just like the *Host*. Its construction is much the same, in that it requires the *Network Layer* to communicate with the *Communications Link* as the *Datalink Layer.* However, the *Traffic Generator* does not require a reliable transmission service, as that is not defined as its role, but it does need to provide traffic stimulus. Hence, it re-uses the *Generator* -- with the construction of a *Network-Adaption Layer* in the same manner as the *Transport-Adaption Layer*. The *Traffic Generator* is shown in Figure 1-2.6.

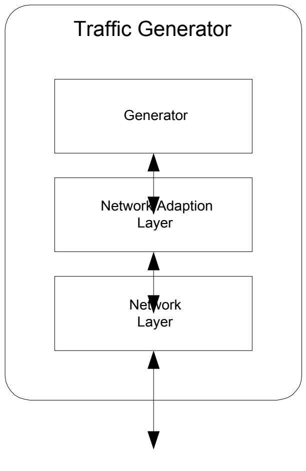


Figure 1-2.6. End System: Traffic Generator

Next, the *Intermediate System* must connect a number *of Communications Links* by way of the *Datalink Layer*. Therefore, it consists of a number of *Network Layers* (as per the OSI Reference Model), which use a *Routing Module* to perform the switching between *Network Layers*. It does not require any other layers. The *Intermediate System* is shown in Figure 1-2.7.

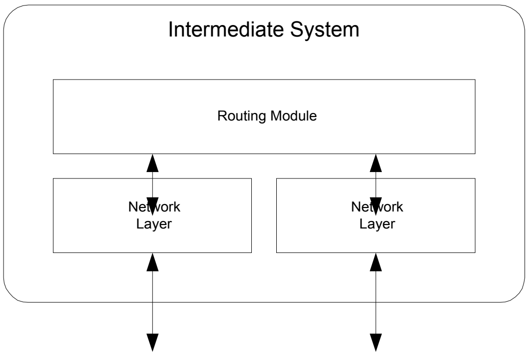


Figure 1-2.7. Intermediate System: Router

Finally, a central control component is required. The OSI architecture does account for this in defining a *Management Information Base (MIB*) that communicates with all layers -- via the Common Management Information Service (CMIS) (Stallings, 1993). This is a single stand-alone entity that has pervasive access to all other entities, defined to be *Management*.

Consider that the Management functionality could be distributed, as an inherent aspect of all other modules, but the decision was made to provide one centralised point at which management occurs, and an "interface" on all entities through which elements are accepted from that central point. *Management* is shown in Figure 1-2.8.

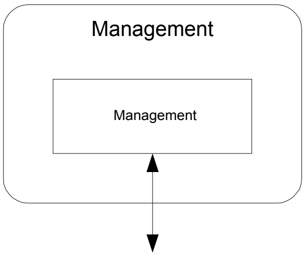


Figure 1-2.8. Management

The extent to which re-useability has been employed was a definite consideration here. The re-useability of the Generator and the Network Layer being prime examples. The use of the adaption layers promotes re-useability (including the case of concept re-usability) and preserves the architectural concepts. Hence, much is gained in this design, with the expense being the construction of two adaption layers -- which are trivial in operation -- and the other cost of genericity in having to maintain consistency and so forth. The gains far outweigh these costs however.

Having established these modules, a communications interface must be examined. In keeping with the OSI Reference Model and the primitives indicated above, all communications between entities will be based upon these primitives--except for the case of the Generator that uses an abstract data type thence mapped by the Adaption Layers. Further, the OSI Reference Model does have, with regard to many layers, a common set of messages used for communication -- Note that message is used in an abstract concept, it may be a remote procedure call, or a data structure, etc. These common messages include:

- **Connect** -- Either a request for a Service User to have the Service Provider establish an operating association, or an indication from the Service Provider to the Service User to notify of an establishment of the operating association (where it was not requested by the Service User). This is local communication.
- **Disconnect** -- Either a request for a Service User to have the Service Provider terminate an operating association, or an indication from the Service Provider to the Service User to notify termination of the operating association (where it was not requested by the Service User). This is local communication.
- **Status** -- An indication of local status information from the Service Provider to the Service User. This is local communication.
- **Data** -- Either a request for a Service User to have the Service Provider transfer an element of data through to the peer Service User, or an indication from the Service Provider to the Service User to notify of received data from the peer Service User. This is peer-to-peer communication in our case, but not always so in the real world.

A simple state diagram, given in Figure 1-2.9, illustrates the basic relationship between these messages.

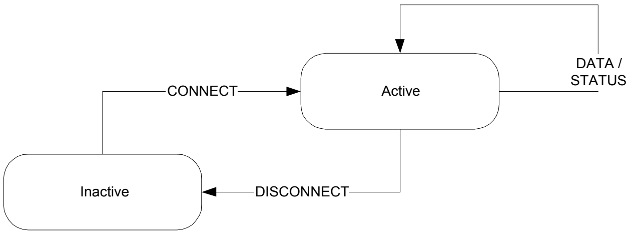


Figure 1-2.9. Message Primitive State Diagram

And, for the purposes of management:

- **Set** -- An indication from Management to have an element of information Set to a specific value in the receiving Layer.

With a message, there are two classes of parameters. The first class is specific to the message at hand, such as length fields, content types, addresses for peer end-points and so on. The second class is an abstract content itself, which is usually of some opaque type--it is merely an encapsulation. To encapsulate information, OSI systems have the concept of an Information Element to represent some unit of information. Therefore, there are two allowed types of encapsulations here: other messages and Information Elements. This relationship is illustrated in Figure 1-2.10.

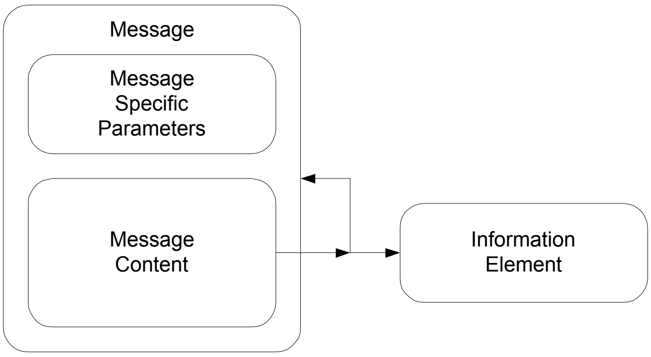


Figure 1-2.10. Message Encapsulation Relationship

This is an ideal situation in the BONeS model. BONeS allows for inheritance in its data hierarchy, and hence the ability to have polymorphic behaviour. In a restriction to two types of encapsulation, the "typing" can be considered to be "strong", and hence subject to better run-time verification measures. Rather than have many lines of communication between the entities, they are aggregated into one single line (in each direction) that can accept a type at the base of the hierarchy for the particular entity/layer. When parsed inside the module, these "base" messages can be promoted back to their original type -- hence, the ability is there to provide more types in communication, yet not affect the external interface and to save unwanted cluttering and potential errors. Figure 1-2.11 illustrates the hierarchy in this model.

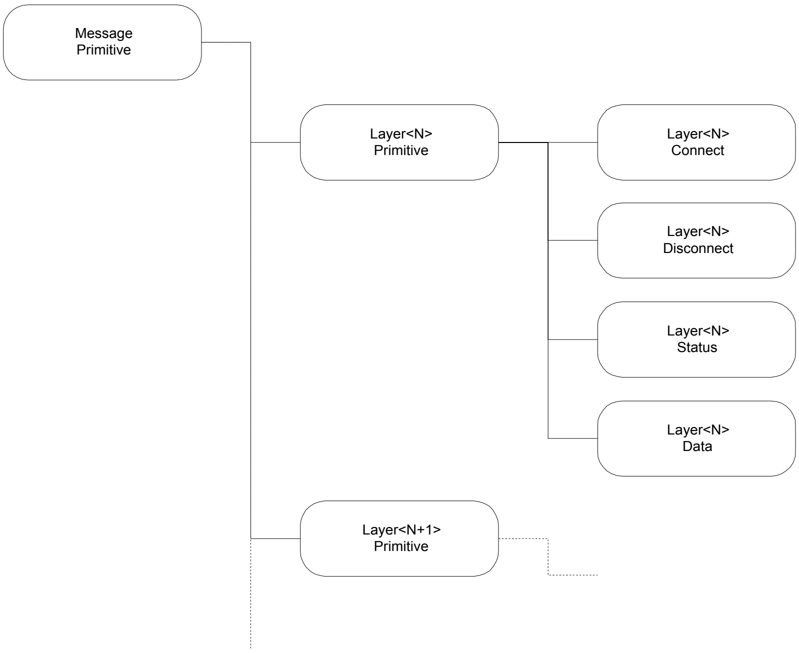


Figure 1-2.11. Message Hierarchy

The architecture must also have the concept of addresses; there are two primary reasons for such. Firstly there is the need for communication between peer components, whether it be connection or connectionless. Thus, all higher-level components (End Systems, Intermediate Systems) have distinct addresses. The next concern is with Management that needs to direct information to a specific component, and - more specifically - an entity within the component. For this case, it is decided that all entities have addresses.

The original high-level design carried out in Thesis 1 constructed an architecture that was not based upon the OSI model. In the intervening time, the realisation was that significant benefit could be derived from an OSI based architecture; so the new architecture was designed. This benefit was seen to require slightly more time in design, but provide benefits in implementation and testing.

2.3. Primary Modules

The significant modules play a major role in the architecture. Detailed design aspects are provided corresponding sections of Appendix 1.

2.3.1. Datalink Layer

2.3.1.1. Overview

The Datalink Layer models a real world communications link. The real world entity manifests itself as a physical transmission line, upon which bits (in some representation) are transported. In transporting these bits, framing may also be used. The link is generally peer-to-peer (either point-to-point, or a multi-drop), but this is not always the case--however, it certainly is for most cases we are concerned with.

The physical nature of transmission lines and their geographical length gives rise to two defining attributes. The first is a Bandwidth: defined to be the number of bits that can be placed onto the link at any given dimension of time (e.g. bits per second). This limits the rate at which the transmission line can pass bits; and hence, information. The second attribute is a Propagation Delay: the result of the bits having to move from one end of the transmission line to the other (fundamentally a result of electron movement).

Both of these attributes result in a delay that is incurred by information traversing the transmission line. The delay due to the Bandwidth is a function of the number of bits passed, whereas the delay due to the Propagation is constant. The total delay incurred can be represented in the following relation:

Delay := Bandwidth (Bits/Sec) / Length (Bits) + Propagation Delay (Sec)

In addition, a third attribute (of sorts) is present: this is more of a derived attribute, as opposed to an inherent one. It is the state of the transmission line: the line can be active or inactive. In the latter state, the line cannot pass information from one end to the other.

An entity using the transmission line places bits, or a collection of bits, onto the line and expects these bits to appear at the peer end. In the real world, examples of a transmission line include coaxial cable, fiber optics and space (for radio spectrum signals).

Note that the physical transmission line described here is represented by Layer 1 and below in the OSI model, however we model at Layer 2. This is because we don't need to model the bit-by-bit transfer, as it is not relevant in a simulation. What we care about is the transmission of a length of structured bits, which are contained in representations of packets, so Layer 2 is sufficient. However, Layer 2 does often provides a reliable Datalink protocol, but we don't consider that case--nevertheless, it is entirely possible to construct such a thing internally without affecting the external interface, a beauty of the OSI model.

The modelling of Layer 2 activity occurs by accepting Datalink Layer messages and acting upon them. Data messages are delayed and thence released to the opposing side's Upper Layer, while Status and Connection messages are generated locally and propagated to the local side's Upper Layer. In delaying Data messages, the Datalink Layer only uses the Length of the message, and no other fields, as per the relation given above.

To model the availability aspect of the line, the Datalink Layer retains state indicating whether or not the line is active or inactive. Data messages that are passed to the layer when the state indicates that it is inactive will be discarded. The Upper Layer is informed of these state changes.

As with other modules, some operations can be carried out via messages from the Management entity. This allows for dynamic behaviour during the execution of a simulation. For the Datalink Layer, the state of the layer can be altered to be either active or inactive. This ability to activate and deactivate a link during the execution of a simulation was considered important.

Internally, the operation of the Datalink Layer is simple. The core operation is the delay of input as per the relation above, but ancillary functionality is required to control the flow of input messages, and perform other control activity.

2.3.1.2. External Interface

The external interface defines this entity as seen by other entities in the architecture and provides a concrete position from which to design the internal construction. The treatment of the external interface requires delineation of the other entities that are communicated with, the elements used for such communication, and the behavioural aspects of the communication.

2.3.1.2.1. Relationships

The Datalink Layer concerns itself with only three external entities. The first two of which are the Upper Layers that represent each end of the modelled data link communications pipe--these directly represent entities in the real world. The third is a Management entity that has been introduced as a means to provoke actions and modify aspects of the Datalink Layer.

The context diagram, in Figure 1-2.12, illustrates the entities and their data relationships.

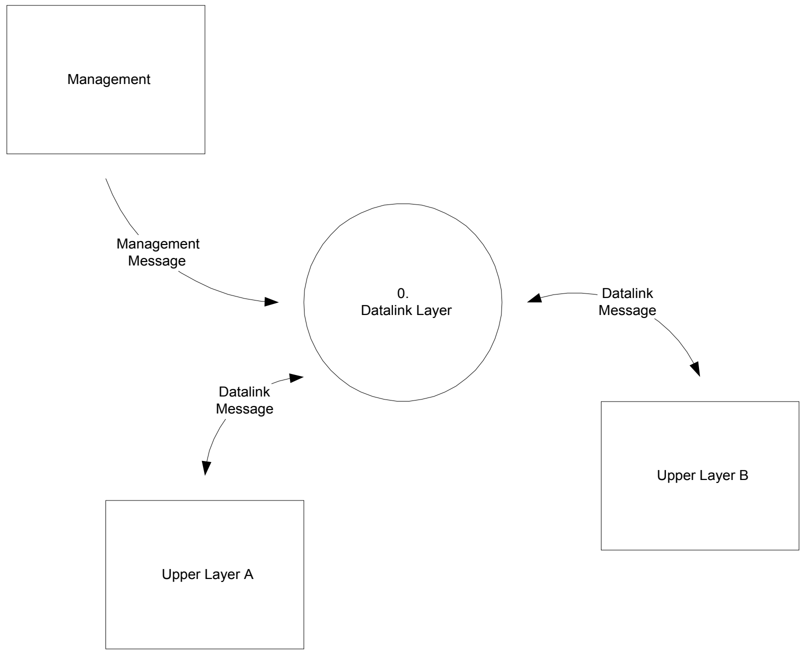


Figure 1-2.12. Datalink Layer: Context Diagram

Table 1-2.1 details the role and types of data communicated.

Name	*Role*	*Communicated Information*
Layer A	One end point for communications pipe	Datalink Messages, for Data and Status transfer
Layer B	The other end point for the communications pipe	Datalink Messages, for Data and Status transfer
Management	Modify behaviour	Management Messages, with Datalink Information Elements

Table 1-2.1. Datalink Layer: Entity Relationships

Table 1-2.2 outlines the content, purpose and description of data communicated.

Name	*Content*	*Purpose*
Datalink Connect Indication Message	Null	Indicate that Datalink is Active
Datalink Disconnect Indication Message	Null	Indicate that Datalink is Inactive
Datalink Data Request Message	Length, Content	Request transfer to alternate end point
Datalink Data Indication Message	Length, Content	Indicate arrival from alternate end point
Datalink Status Indication Message	IE	Indicate Flow Control state
Management Set Indication Message	IE	Modify state
Datalink State IE	Boolean	Change current state
Datalink Flow Control IE	Boolean	Indicate whether flow control released

Table 1-2.2. Datalink Layer: Data Relationships

Any other data that arrives from the external entities to the Datalink Layer is ignored.

2.3.1.2.2. Parameters

Additionally, there are parameters, which define the behaviour of the Datalink Layer. These parameters are represented as data stores internally, but are externally visible and configurable by users of the module. Table 1-2.3 describes these parameters. They must be set to legitimate values for the correct modelling of the Datalink Layer. The *Address* is mandatory for Management to correctly function.

Name	*Purpose*	*Values*	*Default*	*Example*
Address	Allow messages to be directed to this module	Integer	0	10
Bandwidth	The bandwidth in bits per second	Integer	64000	2000000
Propagatn Delay	The propagation delay in seconds	Real	0.007	0.700
State	State of the datalink as being active or inactive	Boolean	True	False

Table 1-2.3. Datalink Layer: Parameters

A design decision was made as to which of these parameters were to be modifiable from the Management entity. A rationale was developed based on whether or not it was realistic for the parameter to alter during the course of a simulation, and whether or not such alteration may be required from simulation perspective.

For the Datalink Layer, the *Bandwidth* and *Delay* characteristics are largely a function of the underlying communications media, which is a fixed and static manifestation (There are cases where it may not be, i.e. a switched call through a telecommunications network). The *State* of a Datalink Layer, however, is more realistically subject to fluctuation, and there are good reasons for why simulations may wish to contrive such fluctuations. Therefore, only *State* is modifiable during simulation execution, whereas *Bandwidth* and *Propagation Delay* remain static.

2.3.1.2.3. Behaviour

- *Datalink Data Request Message* Input

The message will only be processed if the *State* of the Datalink Layer is active (*True*), and if *Flow Control* has been *Released*, indicating that the Datalink Layer is currently not processing an existing message. If either of these conditions are not met, then the message is discarded.

Next, *Flow Control* is *Asserted* and the message is delayed for a time period corresponding to its transmission at the specified *Bandwidth*: using the *Length* of the message to determine this.

When that delay is complete, a *Datalink Status Indication Message* is sent back to the originator. It has a *Datalink Flow Control IE* as its content, indicating that the originator may now transmit another message (i.e. *Flow Control* is now *Released*). The message is then delayed for a time period corresponding to the specific *Propagation Delay*.

In between, and after, each of these delays, the *State* of the Datalink Layer is examined, and the message is discarded if the state is inactive (*False*). Once the final delay is complete, the message is converted into a *Datalink Data Indication Message* and output to the alternate upper layer from which it originated.

- *Datalink Data Indication Message* Output

This output is generated as a result of the process of accepting *a Datalink Data Request Message* at the other end of the Datalink Layer.

- *Datalink Connect Indication Message* Output

This output is generated when the Datalink Layer becomes active. It is a notification that the Datalink Layer is capable of accepting and transporting messages.

- *Datalink Disconnect Indication Message* Output

This output is generated when the Datalink Layer becomes inactive. It is an notification that the Datalink Layer is not capable of accepting and transporting messages.

- *Datalink Status Indication Message* Output

This output is generated when *Flow Control* is released as an indication that the next *Datalink Data Request Message* will be accepted by the Datalink Layer.

- *Management Set Indication Message* Input

The message is first verified to ensure that its Destination Address corresponds to the Datalink Layer's *Address*. If it is not destined for this *Address*, then it is discarded. If accepted, the Information Element in the message is extracted and processed according to its type:

    - *Datalink State IE* -- The current state of the Datalink Layer is altered to that as specified in the IE. If the state becomes active, then a *Datalink Connect Indication Message* is sent to both Upper Layers, otherwise if the state becomes inactive, *a Datalink Disconnect Indication Message* is sent.

2.3.1.2.4. Data Accessors

For constructing and deconstructing data that flows into and out of the Datalink Layer, a number of accessors are designed to encapsulate the direct access to the data structures. Some of these accessors are private, whilst others are public and available to the external users of the Datalink Layer.

Name
Construct Message Datalink Data Request
Construct Message Datalink Connect Indication
Construct Message Datalink Disconnect Indication
Construct Message Datalink Status Indication
Extract Message Datalink Data Indication
Extract Message Datalink Connect Indication
Extract Message Datalink Disconnect Indication
Extract Message Datalink Status Indication
Convert Message Datalink Data Request to Indication
Construct IE Datalink Flow Control
Extract IE Datalink Flow Control
Construct IE Datalink State
Extract IE Datalink State

2.3.1.2.5. Dependencies

The Datalink Layer requires the use of external modules.

Name
Extract Message Management Set Indication

2.3.1.2.6. Initialisation

When the Datalink Layer is first initialised, it will generate a *Datalink Connect Indication Message* or a *Datalink Disconnect Indication Message* depending on the initial value of the *State* parameter. This will ensure that Upper Layers are correctly aware of the Datalink Layer's *State*.

2.3.1.3. Internal Design

2.3.1.3.1. Approach

The approach for the internal design was to partition the functionality two ways:

- *Transmission Channel* as the means by which Datalink messages are relayed between two upper layer peers. This models the acceptance, verification, delay and emergence of the *Datalink Data Request/Indication*

Message and the generation of a *Datalink Status Indication Message* for Flow Control purposes. Parameters are used from data stores to control this behaviour.

- *Management Processor* as an entity that accepts and acts upon messages from Management, and thence sets appropriate local parameters accordingly.

Note that there are two instances of a *Transmission Channel*, and only one instance of a *Management Processor*, with the former using the same parameters. An extension could be to provide an asymmetric capability in terms of *Bandwidth* and *Propagation Delay* characteristics. This architecture is shown in Figure 1-2.13.

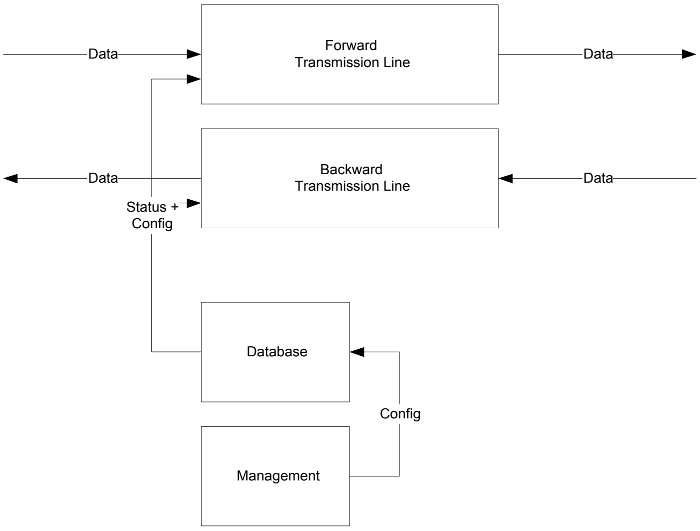


Figure 1-2.13. Datalink Layer: Architecture

2.3.1.3.2. Data Flow Diagrams and Process Specifications

Detailed design information is provided in Appendix 1.

2.3.1.4. Additional notes

The following issues were addressed in the design.

- Possible additional control via Management -- The construction of management processing allows for new functionality to be added with minimal disruption to the existing design.

- Possible asymmetric transmission channels -- The partitioning of the transmission channels and the way in which they use their behavioural information (Bandwidth and Propagation Delay) means that changes to support asymmetric channels are trivial.

- Possible internal buffering of messages -- The flow control mechanism, as externally visible, does not preclude an internal ability to buffer messages for transmission. This, such a change could be carried out and implemented without needing to alter external modules.

- Use of primitive accessors -- At all times, accessors are used to manipulate data structures.
- Performance considerations -- For performance, the flow of data within the transmission channels was designed to be void of any significant processing operations.

2.3.2. Network Layer

2.3.2.1. Overview

A Network Layer is not just a concept, but an actual operating entity in real world communication systems. This layer provides an unreliable datagram oriented service between distant peers. A key feature of the Network Layer, represented as Layer 3 in the OSI Reference Model (International Organisation for Standardisation, 1984), is the ability to address messages to other Network Layers and have them routed within the network, traversing differing underlying transmission media.

The single most important property of the Network Layer is its independence from the underlying transmission media; this is in contrast to the Datalink Layer which is generally more closely tied to the nature of the transmission media. In our architecture, the split between the Datalink Layer and the Network Layer (rather than combining the two into a single entity) allows for the seamless use of differing Datalink Layers.

The Network Layer (Layer 3) protocols (for example, the Internet Protocol (IP) and the Connectionless Network Protocol (CLNP)) also provide specific measures related to the fact they are routed across multiple links. This includes mechanisms to detect infinite looping (Hop Count and Time to Live fields), addresses and congestion related information. The Network Layer is also generally the place that queuing occurs in communications systems, it generally does not occur above the Network Layer, is always provided at the Network Layer, and sometimes is provided at the Datalink Layer. It is within this queuing that messages are lost due to the limited link to which it is connected, because the queue has a finite size and can only transmit items at a finite rate.

Internally, the Network Layer is reasonably trivial. It consists of the core queuing activity, surrounded by ancillary matters to deal with state and layer connection. The queuing activity can be simple, as in a FIFO with overflow, or it can consist of complex policies to dictate how messages are inserted and extracted (in the real world, there may even be a number of queues). Queue length and policies' are specified.

When the Network Layer receives messages from the Upper Layer, it can either transmit them immediately onto the Datalink Layer, or queue them if it is currently in the process of transmitting other messages. Messages received from the Datalink Layer are passed to the Upper Layer immediately if they contain data, or are used for control purposes if they indicate the status of the Datalink Layer.

2.3.2.2. External Interface

The external interface defines this entity as seen by other entities in the architecture and provides a concrete position from which to design the internal construction. The treatment of the external interface requires delineation of the other entities that are communicated with, the elements used for such communication, and the behavioural aspects of the communication.

2.3.2.2.1. Relationships

The Network Layer communicates with two other entities. Unlike other modules, there is no communication with Management. The reason for this is that there were deemed to be no requirements for Management control, however additional in a modified design would be trivial.

The first entity that the Network Layer communicates with is an Upper Layer using Network Messages. The third entity is a Datalink Layer that is used for the eventual transmission of Network Messages. Status information is propagated up from the Datalink Layer, and at the same time, the Network Layer also propagates status information (not necessarily because of what has occurred at the Datalink Layer) up to the Upper Layer.

Figure 1-2.14 illustrates the entities and their data relationships.

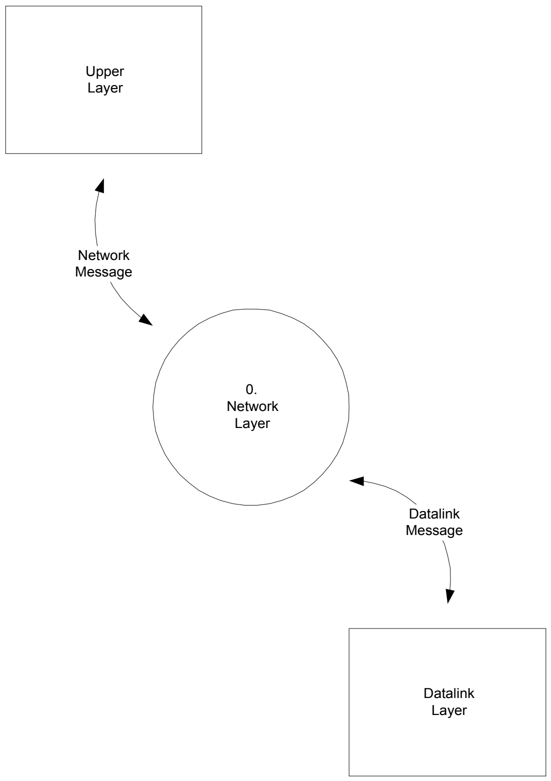


Figure 1-2.14. Network Layer: Context Diagram

Table 1-2.4 details the roles and information communicated between entities.

Name	*Role*	*Communicated Information*
Upper Layer	Uses Network Layer as a delivery agent for Data	Network Messages, for Status and Data transfer
Datalink Layer	Acts as the delivery service for the Network Layer	Datalink Messages, Status indications and Data transfer

Table 1-2.4. Network Layer: Data Relationships

Table 1-2.5 outlines the data.

Name	*Content*	*Purpose*
Network Connect Indication Message	Null	Indicate that Network Layer is active and can deliver messages
Network Disconnect Indication Message	Null	Indicate that Network Layer is inactive and can't deliver
Network Status Indication Message	IE	Indicate Load of Network Layer
Network Data Request Message	Length, Content	Request for Network Layer to deliver messages
Network Data Indication Message	Length, Content	Indicate that message has arrived from peer Network Layer
Datalink Connect Indication Message	Null	Indicates that Datalink Layer is active and can deliver
Datalink Disconnect Indication Message	Null	Indicates that Datalink Layer is inactive and can't deliver
Datalink Status Indication Message	IE	Contains flow control or other IE
Datalink Data Request Message	Length, Content	Encapsulates Network message for delivery via Datalink Layer
Datalink Data Indication Message	Length, Content	Indicates reception of encapsulated Network Message
Datalink Flow Control IE	Boolean	Indicates that flow control is released, so new message can be sent
Network Load IE	Real	Indicates the load on the current Network Layer

Table 1-2.5. Network Layer: Data Relationships

Any other data that arrives from external entities to the Network Layer is ignored.

2.3.2.2.2. Parameters

There are several external parameters that are important for the operation of the Network Layer. Note that in this case, the *Address* is not entirely important for Management operation, but for the purpose of addressing of Network messages. The parameters are outlined in Table 1-2.6.

Name	*Purpose*	*Values*	*Default*	*Example*
Address	Allow messages to be directed to this module	Integer	0	10
End System	Indicate whether this module is part of an end system	Boolean	True	False
Queue Discipline	Specify the input and output policies for the queuing mechanisms	String	"DropTail"	"RED"
Queue Length	Number of messages that the queue can hold.	Integer	20	3

Table 1-2.6. Network Layer: Parameters

2.3.2.2.3. Behaviour

- *Network Data Indication Message* Output

This message is output as the result of two cases. 1) It arrived from the Datalink Layer encapsulated in a Datalink Data message, and either this isn't an *End System*, or this is an *End System* and it has our *Address* in it. 2) This is not an *End System* and the Network Layer because inactive due to the Datalink Layer indicating it was not active any more. These messages are the result of the queue in the Network Layer being emptied back upwards.

- *Network Connect Indication Message* Output

This message is generated for, and sent to, the Upper Layer to indicate that the Network Layer is active and able to receive messages from the Upper Layer. It occurs in response to a *Datalink Connect Indication Message* from the Datalink Layer.

- *Network Disconnect Indication Message* Output

This message is generated for, and sent to, the Upper Layer to indicate that the Network Layer is inactive and unable to receive messages from the Upper Layer. It occurs in response to a *Datalink Disconnect Indication Message* from the Datalink Layer.

- *Network Status Indication Message* Output

This message is generated for, and sent to, the Upper Layer to provide status information about the Network Layer. There is currently only one item that can be provided, and that is the load factor on the Network Layer's queue. This is given in a *Network Load IE* and contains a number normalised to be between 0 and 1.

- *Network Data Request Message* Input

When the Upper Layer sends this message, it is a request for the Network Layer to transport it to Datalink Layer and thence onto another Network Layer. The Network Layer processes this message by either sending it directly to the Datalink Layer after encapsulating it in a *Datalink Data Request Message* or temporarily queuing it before it is sent to the Datalink Layer.

The queuing may involve specific queuing disciplines with regard to how the message is inserted into the queue, and how the message is removed from the queue. When it is removed, it is sent to the Datalink Layer as a *Datalink Data Request Message*.

- *Datalink Data Indication Message* Input

This message arrives from the Datalink Layer. Its content is extracted, and if it is a *Network Data Request Message* then it is first checked to see whether or not it is destined for this *Address* if this is an *End System*. The *Network Data Request Message* is then converted into a *Network Data Indication Message* and sent up to the Upper Layer.

- *Datalink Connect Indication Message* Input

When this message is received, it is a notification that the Datalink Layer is active. From this point onwards, until the reception of a *Datalink Disconnect Indication Message*, the Network Layer can transmit messages to the Datalink Layer. The reception of this message also causes the Outbound Queue to be initialised, and a *Network Connect Indication Message* to be sent to the Upper Layer to inform it of our state.

- *Datalink Disconnect Indication Message* Input

When this message is received, it is a notification that the Datalink Layer is inactive. From this point onwards, until the reception of a *Datalink Connect Indication Message*, the Network Layer cannot transmit messages to the Datalink Layer. The reception of this message also causes the Outbound Queue to be cleared (resulting in the currently queued messages to be flushed to the Upper Layer if this is not an *End System*), and a *Network Disconnect Indication Message* to be sent to the Upper Layer to inform it of current state.

- *Datalink Status Indication Message* Input

This message arrives from the Datalink Layer. If it contains a *Datalink Flow Control IE* indicating *Release* then the Outbound Processing is requested to release the next queued message.

- *Datalink Data Request Message* Output

This output occurs as a result of processing a *Network Data Request Message* from the Upper Layer (either directly, or after being queued). The request is encapsulated within this *Datalink Data Request Message* and sent to the Datalink Layer.

2.3.2.2.4. Data Accessors

There are several items of data that are specific to the Network Layer and hence have data accessors constructed for them.

Name
Construct Message Network Data Request
Construct Message Network Connect Indication
Construct Message Network Disconnect Indication
Construct Message Network Status Indication
Extract Message Network Data Indication
Extract Message Network Connect Indication
Extract Message Network Disconnect Indication
Extract Message Network Status Indication
Convert Message Network Data Request to Indication
Convert Message Network Data Indication to Request
Construct IE Network Load
Extract IE Network Load

2.3.2.2.5. Dependencies

Due to the use of the Datalink Layer entity, modules are required by the Network Layer.

Name
Extract Message Datalink Connect Indication
Extract Message Datalink Disconnect Indication
Extract Message Datalink Status Indication
Extract Message Datalink Data Indication
Construct Message Datalink Data Request
Extract IE Datalink Flow Control

2.3.2.2.6. Initialisation

The initial state of the Network Layer is that it is presumed that the Datalink Layer is not active.

2.3.2.3. Internal Design

2.3.2.3.1. Approach

Internally, there are two main processing blocks in the architecture:

- *Inbound Processing* to process messages that arrive from the Datalink Layer. This involves determining the type of message, and thence acting upon it and its content. So far, this involves propagating state information and altering the outgoing queue's behaviour.
- *Outbound Processing* to process *Network Data Request Messages* that arrived from the Upper Layer and are destined to be delivered to the Datalink Layer--noting that processing involves possibly queuing the Request until the Datalink Layer is able to accept it.

There is no management block, as currently the Network Layer does not have any functionality relating to Management. The functional groupings are shown in the architectural diagram in Figure 1-2.15.

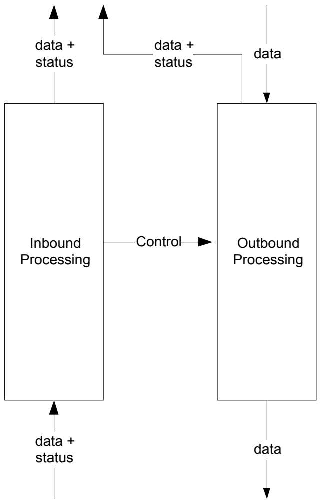


Figure 1-2.15. Network Layer: Architecture

2.3.2.3.2. Data Flow Diagrams and Process Specifications

Detailed design information is provided in Appendix 1.

2.3.2.4. Additional notes

The following issues were addressed in the design.

- Possible additional interpretation of Datalink Messages -- The partitioning of processing for Datalink Messages is such that modification and/or addition of new functionality is trivial.
- Consistent architectural split -- The same architectural split has been employed as with most other modules; in terms of input, output and management delineation.
- Use of type switching for Messages and IEs -- The switches that classify Network Layer Messages and Information Elements are such that they can be used on hierarchically typed data structures, an inherent ability for BONeS.
- Use of primitive accessors -- At all times, accessors are used to manipulate data structures.
- Placement of simulation Probes -- A number of stubs have been put into place to facilitate points at which Probes can be added for the purposes of collecting data during simulations.
- Additional queue input and output disciplines -- The input and output disciplines are designed in such a way that it is trivial to add further. In the case of an input policy, this would be an alternate selection, whereas for output, it can be either alternate or in tandem with other disciplines.
- Replaced outgoing processing -- For even more radical modifications, the outgoing processing functionality is segmented to the extent that it could be replaced entirely without affecting surrounding processing significantly.

- End and Intermediate System classification -- To facilitate re-use of this module, the End system parameter allows for the Network Layer to be used in an End System which does have concern about Addresses, or otherwise an Intermediate System which does not have concern about Addresses.

- Performance of queue ADT -- The queue was designed as an ADT partially due to anticipation of it being implemented in a primitive language ('C') for reasons of performance.

2.3.3. Transport Layer

2.3.3.1. Overview

Real world communications systems do often have Transport Layers. This layer provides a reliable delivery service between two specified end points. They expect to have an unreliable transport medium--i.e. a Network Layer--at their disposal. Upon this, the Transport Layer builds a reliable service by being able to detect lost data and then retransmit that data until reception occurs. At its upper boundary, the Layer is often stream oriented, in that it does not honour data boundaries between end-points, but views all data as contiguous--hence, a receiver can not expect to receive transport data messages with the same boundaries that were sent. Transport connections are also, apart from as a research experiments, point-to-point and addressed.

Two well-known Transport Protocols are the ISO Transport Protocol 4 (TP4). and the DARPA Transmission Control Protocol (TCP) (RFC793, 1981). The latter is under study in this work. TCP is a complex protocol specified as a set of states, the conditions for transition between those states, and legitimate behaviour that can occur within those states. Two significant divides in these states are those concerned with the establishment and termination of connections, and that concerned with the transfer of data on an active session.

To transport data, TCP implements a sliding window based protocol. For a transmitter, this means that the successful transfer of data is indicated by the reception of acknowledgments from the receiver, allowing for more data to be sent. The receiver is able to adjust window sizes in order to ensure that the transmitter does not have too much data in transit at any given point in time. There are additional mechanisms such as those for congestion avoidance and control, (re)transmission timeouts, silly window syndromes, round trip time estimation and so on.

Our model of the Transport Layer consists of providing the stream oriented transport service to an Upper Layer, allowing it to indicate a number of octets that it requires to be transported. The Upper Layer can also initiate and terminate connections. The core Transport Layer functionality is carried out by the Transmission Control Protocol (TCP), which is modelled upon the BSD4.4/Net3 TCP implementation. Key modelling aspects are:

- The removal of all but ESTABLISHED state processing, as the core data transfer is the only functionality we are interested in.
- The need for fragment queues and data processing, but the model is only concerned with data lengths, so therefore does not hold or process data per se.
- Removal of urgent data processing, as it is not used here.
- Removal of most options processing, as it is not used here.

For a detailed explanation of TCP, (Stevens, 1995) should be consulted. Such a detailed explanation would consume considerable space here.

The model is also concerned with compartmentalising and abstracting the TCP processing to be that of a generic Transport Layer. A significant advantage of this is that other transport protocols can be inserted without modification to the external activity of the Transport Layer, but also, testing is assisted.

Because we only retain TCP's ESTABLISHED processing, there needs to be some way for the two end points involved in a conversation to synchronise themselves. The only aspect of synchronisation is the Initial Sequence Number, and it is this which can be set via the Management entity. Obviously, the entity must set both this at both end points. The peer Address along with connect and disconnect operations originate from the Upper Layer via transport messages.

2.3.3.2. External Interface

The external interface defines this entity as seen by other entities in the architecture and provides a concrete position from which to design the internal construction. The treatment of the external interface requires delineation of the other entities that are communicated with, the elements used for such communication, and the behavioural aspects of the communication.

2.3.3.2.1. Relationships

The Transport Layer is in communication with three entities. The first one of which is the Management entity, as is the case with many other modules. Management messages originate from this entity and are used to modify the Transport Layer internally. The second entity is an Upper Layer that communicates via. Transport Layer specific messages. The Upper Layer expects specific functionality to occur from using these messages. In addition, the reception of messages from the Transport Layer is known to occur under specific circumstances. The third entity is a Network Layer, with which Network Messages are communicated. The Transport Layer expects to transmit and receive Data Messages via. the Network Layer and is capable of processing data and status messages propagated up from the Network Layer.

Figure 1-2.16 illustrates the entities and their data relationships.

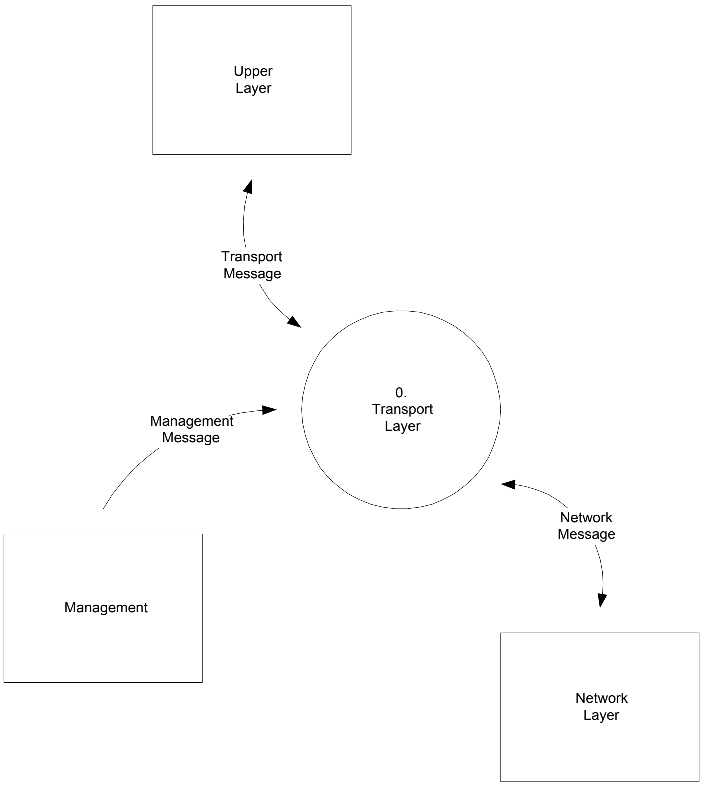


Figure 1-2.16. Transport Layer: Context Diagram

Table 1-2.7 details the roles and information communicated between entities.

Name	*Role*	*Communicated Information*
Upper Layer	Requests start, stop and transfer of information on	Transport Layer Messages
Network Layer	Acts as the delivery agent for the Transport Layer	Network Layer Messages, Status and Data information
Management	Modify behaviour of Transport Layer	Management Messages, with IEs

Table 1-2.7. Transport Layer: Entity Relationships

Table 1-2.8 outlines the data.

Name	*Content*	*Purpose*
Transport Connect Request Message	Address	Requests the connection of a session to the given address
Transport Disconnect Request Message	Null	Requests the disconnect of a currently connected session
Transport Data Request Message	Length	Requests transfer of Data on the current session
Transport Data Indication Message	Length	Indicates transfer of data on the current session
Network Connect Indication Message	Null	Indicates that Network Layer can accept messages
Network Disconnect Indication Message	Null	Indicates that the Network Layer cannot accept messages
Network Data Request Message	Length, Content	Requests transfer of information (encapsulated Transport) via Network Layer
Network Data Indication Message	Length, Content	Indicates the transfer of information (encapsulated Transport) via Network Layer
Network Status Indication Message	IE	Indicates status information about the Network Layer
Management Set Indication Message	IE	Modifies behaviour of Transport Layer
Transport Setup IE	ISN	Conveys Initial Sequence Number for start of Transport sessions

Table 1-2.8. Transport Layer: Data Relationships

Any other data that arrives from external entities to the Transport Layer is ignored.

2.3.3.2.2. Parameters

Apart from the mandatory *Address*, there are no externally visible parameters for the Transport Layer. However, there are some internal parameters, which are directly affected externally--through Management and other messages--that warrant mention. The parameters are shown in Table 1-2.9.

Name	*Purpose*	*Values*	*Default*	*Example*
Address	Allow messages to be directed to this module	Integer	0	10
Initial Sequence Number	Allow for end point synchronisation	Integer	0	13214
Dest Address	Specify the end point for the communication	Integer	0	10

Table 1-2.9. Transport Layer: Parameters

Some discussion is required as to the reason for allowing the *Initial Sequence Number* to be set via Management. In this model of TCP, we have only kept the TCP Established processing, and not concerned ourselves with the opening and closing synchronisation states. The reason for this was to first remove complexity, and also (more importantly) that these states were not needed for the simulation scenarios that we envisaged--and, such initial synchronisation would cloud the real issues that we are concerned with. However, having made that decision, the case still arises as to how two peers in a session do perform synchronisation (to establish an Initial Sequence Number). This was solved by allowing Management to configure both peers' *Initial Sequence Number*. The same number will be used for each new Session, so it is possible to change it for every subsequent session, or leave it at a single value.

2.3.3.2.3. Behaviour

- *Network Data Indication Message* Input

This message arrives from the Network Layer. When received, the content of the message is extracted, but only if the current state indicates that the Transport Session is active. This content consists of a *TCP Packet* which is passed into *TCP Processing* to be dealt with by a TCP Input process.

TCP Input processing involves core TCP functionality; such as verifying that the packet is valid by way of the current sequence number and known window positions, thence extracting out the data in the packet and passing it to the Upper Layer or internally storing it in the case of out of order arrivals. Acknowledgments are also processed, and this may result in the generation of TCP packets to be passed back down to the Network Layer for transmission to the session's peer.

TCP processing is fairly detailed. An explanation of it will be given when the core functionality is treated specifically.

- *Network Connect Indication Message* Input

This message arrives from the Network Layer. There is currently no defined processing for it, so the message is ignored. A stub module is provided for the addition of processing, if needed.

- *Network Disconnect Indication Message* Input

This message arrives from the Network Layer. There is currently no defined processing for it, so the message is ignored. A stub module is provided for the addition of processing, if needed.

- *Network Status Indication Message* Input

This message arrives from the Network Layer. There is currently no defined processing for it, so the message is ignored. A stub module is provided for the addition of processing, if needed.

- *Network Data Request Message* Output

This output occurs as a result of a generated *TCP Packet* that is required to be passed to the session's peer. The Transport Layer encapsulates the *TCP Packet* into this message and adds the appropriate *Destination Address* before passing it to the Network Layer. It expects the Network Layer to, or at least attempt to, deliver this message to the specified address.

- *Management Set Indication Message* Input

The message is first verified to ensure that its Destination Address corresponds to the Transport Layer's *Address.* If it is not destined for this *Address*, then it is discarded. If accepted, the Information Element in the message is extracted and processed according to its type:

  - *Transport Setup IE* -- The contents of this IE is an *Initial Sequence Number* to be used for the start of the TCP Session's processing. The content is extracted and stored for use on subsequent session starts.

- *Transport Data Request Message* Input

This message arrives from the Upper Layer. When received, the content of the message is extracted, but only if the current state indicates that the Transport Session is active. This content consists of *Data to TCP*, which is passed into *TCP Processing* to be dealt with by a TCP Output process.

TCP Output processing involves core TCP functionality; such as determining whether a packet can be sent due to the current window constraints, thence constructing and sending the data as a *TCP Packet* via. the Network Layer for transmission to the session's peer.

TCP processing is fairly detailed. An explanation of it will be given when the core functionality is treated specifically.

- *Transport Connect Request Message* Input

This message arrives from the Upper Layer. It is an indication that the Upper Layer desires the establishment of a Transport Session to a peer *Destination Address* that is also supplied in the message. Upon receiving this message, the Transport Layer will configure the *TCP Processing* and set its state accordingly. From which point onwards, Data messages can be sent or received by the Transport Layer, until a Disconnect occurs.

- *Transport Disconnect Request Message* Input

This message arrives from the Upper Layer. It is an indication that the Upper Layer desires the termination of a currently active Transport Session. As such, the Transport Layer will update its state accordingly, and terminate all *TCP Processing.*

- *Transport Data Indication Message* Output

This output is delivered to the Upper Layer and occurs as a result of Data becoming available from *TCP Processing*. The Data is represented by its length, and is encapsulated within the message and passed to the Upper Layer.

2.3.3.2.4. Data Accessors

Data Accessors are required for access to both the Transport Layer Messages and Information Elements.

Name
Construct Message Transport Connect Request
Construct Message Transport Disconnect Request
Construct Message Transport Data Request
Construct Message Transport Data Indication
Extract Message Transport Connect Request
Extract Message Transport Disconnect Request
Extract Message Transport Data Request
Extract Message Transport Data Indication
Construct IE Datalink Flow Control
Extract IE Datalink Flow Control
Construct IE Transport Setup
Extract IE Transport Setup

2.3.3.2.5. Dependencies

Due to the use of the Management and Network Layer entities, other modules are required by the Transport Layer.

Name
Extract Message Management Set Indication
Construct Message Network Data Request
Construct Message Application Data
Extract Message Application Data
Extract Message Network Connect Indication
Extract Message Network Disconnect Indication
Extract Message Network Status Indication
Extract Message Network Data Indication

2.3.3.2.6. Initialisation

The initial state of the Transport Layer is that it is presumed to not be active. In addition, the *Initial Sequence Number* is set to zero. The *Address* of the Transport Layer must be defined if any Management Messages are to be received.

2.3.3.3. Internal Design

2.3.3.3.1. Approach

This module is perhaps the most complex of all modules due to the inherent complexity of the Transmission Control Protocol (TCP). As such, the main design concern was to compartmentalise the complex functionality around more simplistic functionality--i.e. to firewall. There are three main architectural groups--status and connection processing, management with database storage and data processing. These are specifically divided into the following blocks:

- *Connection Manager* to process status, connect and disconnect messages from both the Upper Layer and from the Network Layer. This includes setting up and clearing a Transport Session.

- *Management Processing* to process messages that arrive from Management--currently, this only affects the *Initial Sequence Number*.
- *Transmission Control Protocol Processor* as the block within with the TCP protocol is executed. This block is separated from other blocks in such a manner that any transport protocol can be placed into here.
- *Transport Interface* to provide the interface between the transport protocol specific (TCP) processing, and the Upper Layer. This consists of shuttling data between itself and the transport protocol (i.e. TCP).
- *Network Interface* to process arrived Data messages from the Network Layer and departing Data messages for the Network Layer. This consists of extracting and encapsulating (respectively) the transport protocol (i.e. TCP) specific information out of, or in to, Network Layer messages.

The two interfaces were purposely developed to hide Upper and Lower Layer specifics from the transport protocol. This allows for the transport protocol to be developed as a fairly separate entity and, further, to facilitate drop in replacement of other transport protocols: without affecting external modules. These functional groupings are shown in the architectural diagram in Figure 1-2.17.

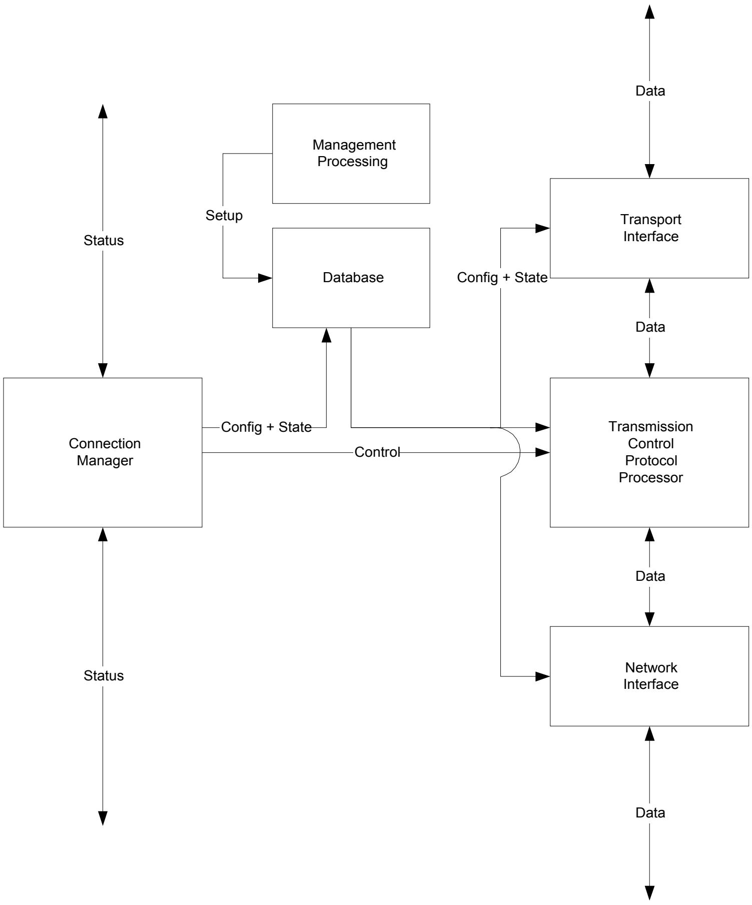


Figure 1-2.17. Transport Layer: Architecture

2.3.3.3.2. Data Flow Diagrams and Process Specifications

Detailed design information is given in Appendix 1.

2.3.3.4. Additional notes

The following issues were addressed in the design.

- Possible additional interpretation of Network Messages -- The partitioning of processing for Network Messages is such that modification and/or addition of new functionality is trivial.
- Use of type switching for Messages and IEs -- The switches that classify Messages and Information Elements are such that they can be used on hierarchically typed data structures, an inherent ability for BONeS.
- Use of primitive accessors -- At all times, accessors are used to manipulate Data Structures.
- Placement of simulation Probes -- A number of stubs have been put into place to facilitate points at which Probes can be added for the purposes of collecting data during simulations.
- Separation of core Transport Protocol -- The Transport Protocol in use has been neatly confined within a processing block in the Transport Layer; allowing for it to be replaced with other type of Transport Protocol.
- Consideration of the TCP Protocol as a separate entity -- For the purposes of risk management, and implementation flexibility, the TCP Protocol is a separate entity.

2.3.4. Network-Adaption Layer

2.3.4.1. Overview

The Network-Adaption Layer is provided as a generative module for the purposes of supporting our simulation architecture--it does not model an entity in the real world, but exists to support our models of the real world.

Its role is to act as a bridge between an Upper Layer (usually a Generator) and the Network Layer. There is one service provided to the Upper Layer, and that is the transfer of a data message of specified length, via. the Network Layer (using a Data Request). The Upper Layer is defined to be dumb, in that it has no knowledge of Networks and the suchlike -- this is a good separation of concerns. Hence, the Network-Adaption Layer contains and supplies the Addresses to be used in the generation of Network Layer Data Requests.

Because the Network-Adaption Layer is used in situations where arbitrary data messages are supplied to the network, and due to specific requirements for our simulations, a list of Addresses can be specified. A single Address is selected randomly from the list for each Data Request that is constructed. This allows for the Network-Adaption Layer to be used in the construction of a composite entity that can "spray" Data Requests to various destinations in a random fashion--exactly what we need for generating background traffic in our simulations.

As with other modules, configuration can be carried out by way of the Management entity. The only parameter of concern in this module is the just indicated Address List used in Data Request generation. There are two reasons for configuration in this manner. The first is that it is much easier and flexible to configure a list of items from a file rather than having to manually enter them in a static simulation set up. Secondly, there are envisaged situations where during the execution of a simulation, the address list may be required to change to contrive specific conditions.

Internally, the Network-Adaption Layer is simple in construction. It has been left with an open architecture to facilitate expansion, as is the general methodology employed in the construction of all modules. There have been specific considerations given to requirements for simulations in that apparently redundant processing paths are evident as places for probe attachments.

2.3.4.2. External Interface

The external interface defines this entity as seen by other entities in the architecture and provides a concrete position from which to design the internal construction. The treatment of the external interface requires delineation of the other entities that are communicated with, the elements used for such communication, and the behavioural aspects of the communication.

2.3.4.2.1. Relationships

There are three entities that communicate with the Network-Adaption Layer. The first one is, as with most other modules, the Management entity. It is from this that Management Messages originate destined for the Network-Adaption Layer: they perform some kind of operation. The next entity is an Upper Layer that provides an abstract item of Data, this Data is modelled by its Length--there is no need to have *actual* data per se. The third entity is a Network Layer, which the Network-Adaption Layer uses to transmit and receive Data Messages to other network connected peers.

Figure 1-2.18 illustrates the entities and their relationships.

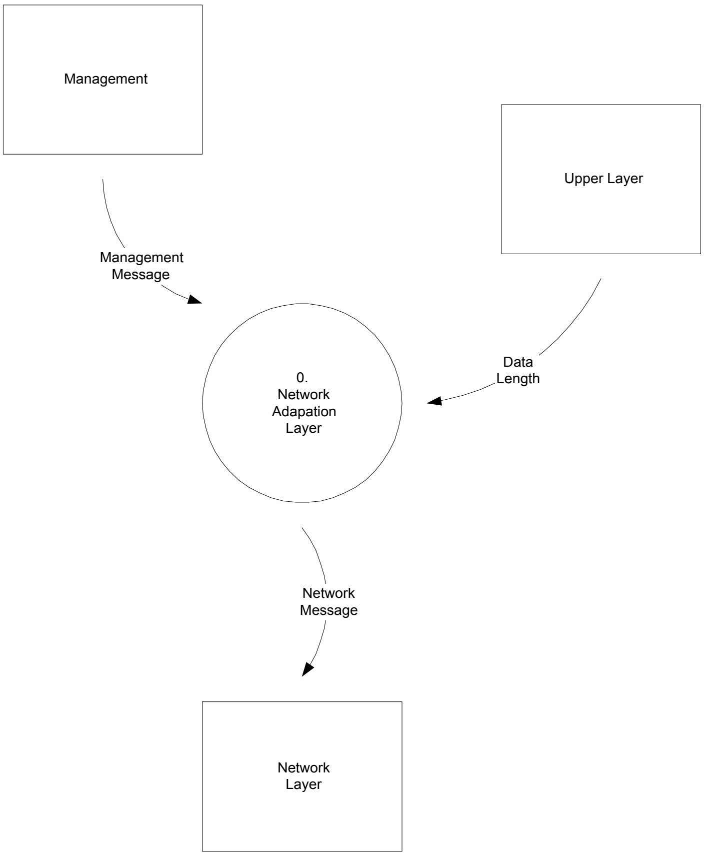


Figure 1-2.18. Network-Adaption Layer: Context Diagram

Table 1-2.10 details the roles and information communicated between entities.

Name	*Role*	*Communicated Information*
Upper Layer	Provides data elements to be transferred via Network Layer	Data Length to be sent
Network Layer	Acts as the delivery agent for the data elements	Network Layer Messages, Data and Status information
Management	Modifies the behaviour of the Network-Adaption Layer	Management Messages, with IEs

Table 1-2.10. Network-Adaption Layer: Entity Relationships

Table 1-2.11 outlines the data.

Name	*Content*	*Purpose*
Network Connect Indication Message	Null	Indicates that Network Layer is able to send messages
Network Disconnect Indication Message	Null	Indicates that Network Layer is unable to send messages
Network Status Indication Message	IE	Indicates status information about Network Layer
Network Data Request Message	Length, Content	Requests transfer of data elements via Network Layer
Network Data Indication Message	Length, Content	Indicates arrival of data elements via Network Layer
Data Length	Integer	Length of data to transfer
Management Set Indication Message	IE	Provides IEs to modify behaviour
Network-Adaption Address List IE	List of Address	Indicates Addresses to be used in delivery of data elements

Table 1-2.11. Network-Adaption Layer: Data Relationships

Any other data that arrives from external entities to the Network-Adaption Layer is ignored.

2.3.4.2.2. Parameters

Apart from the mandatory *Address*, there are no externally visible parameters for the Network-Adaption Layer. However, the *Address List* is an internal parameter that is directly modifiable by Management, therefore it is considered an externally visible parameter--just indirectly accessed. The parameters are shown in Table 1-2.12.

Name	*Purpose*	*Values*	*Default*	*Example*
Address	Allow messages to be directed to this module	Integer	0	10
Address List	Used to destinations for outgoing data messages	Set: Integer	0	1,2,3,4

Table 1-2.12. Network-Adaption Layer: Parameters

2.3.4.2.3. Behaviour

- *Network Data Indication Message* Input

This message arrives from the Network Layer. There is currently no defined processing for it, so the message is ignored. A stub module is provided for the addition of processing, if needed.

- *Network Connect Indication Message* Input

When this message is received, it is a notification that the Network Layer is active. From this point onwards, until the reception of a *Network Disconnect Indication Message*, the Network-Adaption Layer will transmit messages to the Network Layer (when it needs to).

- *Network Disconnect Indication Message* Input

When this message is received, it is a notification that the Network Layer is inactive. From this point onwards, until the reception of a *Network Connect Indication Message*, the Network-Adaption Layer will not transmit any messages to the Network Layer--even if the Upper Layer requests such an action.

- *Network Status Indication Message* Input

This message arrives from the Network Layer. There is currently no defined processing for it, so the message is ignored. A stub module is provided for the addition of processing, if needed.

- *Data Length* Input

The Upper Layer indicates the length of an item of data as a request for the transmission of an element of Data of that length. The Network-Adaption Layer will first check to see whether or not the Network Layer is able to receive messages (i.e. as a result of Connect/Disconnect notifications). If it is, then a *Network Data Request Message* is generated with the *Data Length* and with a random *Address* selected from *the Address List* that was configured by Management. The message is then sent to the Network Layer.

- *Management Set Indication Message* Input

The message is first verified to ensure that its Destination Address corresponds to the Network-Adaption Layer's *Address*. If it is not destined for this *Address*, then it is discarded. If accepted, the Information Element in the message is extracted and processed according to its type:

  - *Network-Adaption Address List IE* -- The contents of this IE are a set of *Addresses* to be used for outgoing *Network Data Request Messages*. When such a message is created, a random *Address* is selected from this list. Note that if only one *Address* is present in the list then it will always be used as the selected *Address*.

- *Network Data Request Message* Output

This output occurs as a result of processing the *Data Length* input from the Upper Layer. The Network-Adaption Layer expects the Network Layer to, or at least attempt to, deliver this message to the *Address* specified in the creation of the message.

2.3.4.2.4. Data Accessors

The only Network-Adaption Layer specific data is the *Network-Adaption Address List IE*, which is generated by Management and processed by the Network-Adaption Layer.

Name
Construct IE Network-Adaption Address List
Extract IE Network-Adaption Address List

2.3.4.2.5. Dependencies

Due to the use of the Management and Network Layer entities are required by the Network-Adaption Layer.

Name
Extract Message Management Set Indication
Construct Message Network Data Request
Construct Message Application Data
Extract Message Network Connect Indication
Extract Message Network Disconnect Indication
Extract Message Network Status Indication
Extract Message Network Data Indication

2.3.4.2.6. Initialisation

The initial state of the Network-Adaption Layer is that it is presumed that the Network Layer is not active. Also, the *Address List* has no entries. The *Address* of the Network-Adaption Layer must be defined if any Management Messages are to be received.

2.3.4.3. Internal Design

2.3.4.3.1. Approach

The internal functionality was divided into the three main processing blocks:

- *Inbound Processing* to process messages that arrive from the Network Layer. This includes classification of the message, and extraction and interpretation of its content--affecting the known state of the Network Layer.
- *Outbound Processing* to process the Data Length request from the Upper Layer by constructing an outgoing *Network Data Request Message* with appropriate Length and randomly selected *Address* from the *Address List.*
- *Management Processing* to process messages that arrive from Management--currently only affecting the *Address List.*

These functional groupings are shown in the architectural diagram in Figure 1-2.19.

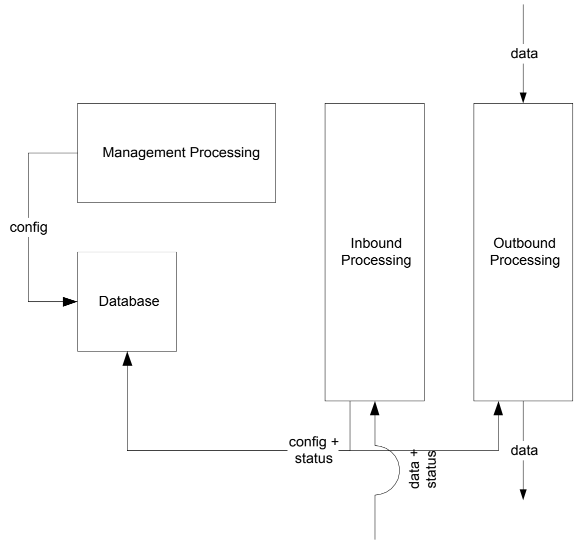


Figure 1-2.19. Network-Adaption Layer: Architecture

2.3.4.3.2. Data Flow Diagrams and Process Specifications

Detailed design information is provided in Appendix 1.

2.3.4.4. Additional notes

The following issues were addressed in the design.

- Possible additional control via Management -- The construction of management processing allows for new functionality to be added with minimal disruption to the existing design.
- Possible additional interpretation of Network Messages -- The inclusion of stub modules allows for the insertion of processing for the Network Layer messages that are currently not examined.
- Use of type switching for Messages and IEs -- The switches that classify Network Layer Messages and Information Elements are such that they can be used on hierarchically typed data structures, an inherent ability for BONeS.
- Consistent architectural split -- The same architectural split has been employed as with most other modules; in terms of input, output and management delineation.
- Use of primitive accessors -- At all times, accessors are used to manipulate data structures.
- Placement of simulation Probes -- A number of stubs have been put into place to facilitate points at which Probes can be added for the purposes of collecting data during simulations.

2.3.5. Transport-Adaption Layer

2.3.5.1. Overview

The Transport-Adaption Layer is provided as a generative module for the purposes of supporting our simulation architecture--it does not model an entity in the real world, but exists to support our models of the real world.

Its role is to act as a bridge between an Upper Layer (usually a Generator) and the Transport Layer. There is one service provided to the Upper Layer, and that is the transfer of a data of a specified length, via the Transport Layer (using a Data Request). The Upper Layer is defined to be dumb, in that it has no knowledge of Networks and the suchlike -- this is a good separation of concerns. The Transport-Adaption Layer must also initiate and terminate Transport Layer associations; hence it is capable of Connecting and Disconnecting Transport Sessions in response to Management requests. When connecting, the Connect Request sent to the Transport Layer will contain the Address for which the Transport Session is peered with.

Internally, the Transport-Adaption Layer is simple in construction. It has been left with an open architecture to facilitate expansion, as is the general methodology employed in the construction of all modules. There have been specific considerations given to requirements for simulations in that apparently redundant processing paths are evident as places for probe attachments.

2.3.5.2. External Interface

The external interface defines this entity as seen by other entities in the architecture and provides a concrete position from which to design the internal construction. The treatment of the external interface requires delineation of the other entities that are communicated with, the elements used for such communication, and the behavioural aspects of the communication.

2.3.5.2.1. Relationships

There are three entities that communicate with the Transport-Adaption Layer. The first one is, as with most other modules, the Management entity. It is from this that Management Messages originate, destined for the Transport-Adaption Layer: they perform some kind of operation. The next entity is an Upper Layer that provides an abstract item of Data, this Data is modelled by its Length--there is no need to have *actual* data per se. The third entity is a Transport Layer, which the Transport-Adaption Layer uses to transmit and receive Data, Connect and Disconnect Messages.

Figure 1-2.20 illustrates the entities and their relationships.

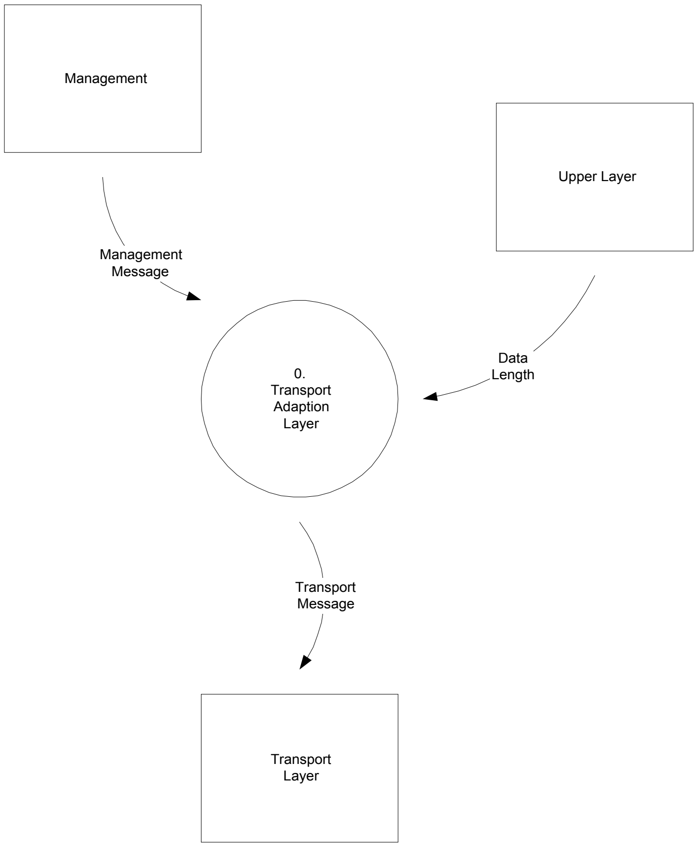


Figure 1-2.20. Transport-Adaption Layer: Context Diagram

Table 1-2.13 details the roles and information communicated between entities.

Name	*Role*	*Communicated Information*
Upper Layer	Provides data elements for transport	Data Length, size of the element
Transport Layer	Acts as the delivery agent for the data elements	Transport Messages, either setup/teardown or data
Management	Modifies behaviour of the Transport-Adaption Layer	Management Messages with IEs

Table 1-2.13. Transport-Adaption Layer: Entity Relationships

Table 1-2.14 outlines the data.

Name	*Content*	*Purpose*
Transport Connect Request Message	Address	Requests establishment of transport session to address
Transport Disconnect Request Message	Null	Requests termination of current transport session
Transport Data Request Message	Length	Requests transfer of length of data on current session
Transport Data Indication Message	Length	Indicates arrival of length of data on current session
Management Set Indication Message	IE	Conveys changes for Transport-Adaption Layer
Transport-Adaption Connect IE	Address	Requests setup to Address to be carried out
Transport-Adaption Disconnect IE	Null	Requests teardown to be carried out
Data Length	Integer	Length of data to be transferred

Table 1-2.14. Transport-Adaption Layer: Data Relationships

Any other data that arrives from external entities to the Transport-Adaption Layer is ignored.

2.3.5.2.2. Parameters

Apart from the mandatory *Address*, there are no externally visible parameters for the Transport-Adaption Layer. The parameters are shown in Table 1-2.15.

Name	*Purpose*	*Values*	*Default*	*Example*
Address	Allow messages to be directed to this module	Integer	0	10

Table 1-2.15. Transport-Adaption Layer: Parameters

2.3.5.2.3. Behaviour

- *Transport Data Indication Message* Input

This message arrives from the Transport Layer. There is currently no defined processing for it, so the message is ignored. A stub module is provided for the addition of processing, if needed.

- *Transport Connect Indication Message* Input

This message arrives from the Transport Layer. There is currently no defined processing for it, so the message is ignored. A stub module is provided for the addition of processing, if needed.

- *Transport Disconnect Indication Message* Input

This message arrives from the Transport Layer. There is currently no defined processing for it, so the message is ignored. A stub module is provided for the addition of processing, if needed.

- *Data Length* Input

The Upper Layer provides a *Data Length* as a request for the transmission of an element of Data of that length. The Transport-Adaption Layer will generate a *Transport Data Request Message* with the specified *Data Length*. The message is then sent to the Transport Layer.

- *Management Set Indication Message* Input

The message is first verified to ensure that its Destination Address corresponds to the Transport-Adaption Layer's *Address*. If it is not destined for this *Address*, then it is discarded. If accepted, the Information Element in the message is extracted and processed according to its type:

  - *Transport-Adaption Connect IE* -- The content of this IE is a single *Address*. This is used in the construction of a *Transport Connect Request Message* that is sent to the Transport Layer as a request for a session to be established to the peer *Address*.
  - *Transport-Adaption Disconnect IE* -- There is no content in this IE. When it is received, a *Transport Disconnect Request Message* will be sent to the Transport Layer as a request for the current session to be terminated.

- *Transport Data Request Message* Output

This output occurs as a result of processing the *Data Length* input from the Upper Layer. The Transport-Adaption Layer expects the Transport Layer to deliver this message via the currently active Transport Session [as configured via. a Connect message].

- *Transport Connect Request Message* Output

This output occurs as a result of processing a *Transport-Adaption Connect IE* received from Management. It instructs the Transport Layer to establish a Transport Session with another Transport Layer at the *Address* specified in the message.

- *Transport Disconnect Request Message* Output

This output occurs as a result of processing a *Transport-Adaption Disconnect IE* received from Management. It instructs the Transport Layer to terminate the current Transport Session.

2.3.5.2.4. Data Accessors

There are two data items that are specific to the Transport-Adaption Layer.

Name
Construct IE Transport-Adaption Connect
Extract IE Transport-Adaption Connect
Construct IE Transport-Adaption Disconnect
Extract IE Transport-Adaption Disconnect

2.3.5.2.5. Dependencies

Due to the use of the Management and Transport Layer entities modules are required by the Transport-Adaption Layer.

Name
Extract Message Management Set Indication
Construct Message Transport Data Request
Construct Message Transport Connect Request
Construct Message Transport Disconnect Request
Extract Message Network Connect Indication
Extract Message Network Disconnect Indication
Extract Message Network Data Indication

2.3.5.2.6. Initialisation

The *Address* of the Transport-Adaption Layer must be defined if any Management Messages are to be received.

2.3.5.3. Internal Design

2.3.5.3.1. Approach

The internal functionality was divided into the three main processing blocks:

- *Inbound Processing* to process messages that arrive from the Transport Layer. This includes classification of the message, and extraction and interpretation of its content.
- *Outbound Processing* to process the Data Length request from the Upper Layer by constructing an outgoing *Transport Data Request Message* of appropriate length.
- *Management Processing* to process messages that arrive from Management--these result in the transmission of Connect and Disconnect Messages to the Transport Layer.

These functional groupings are shown in the architectural diagram in Figure 1-2.21.

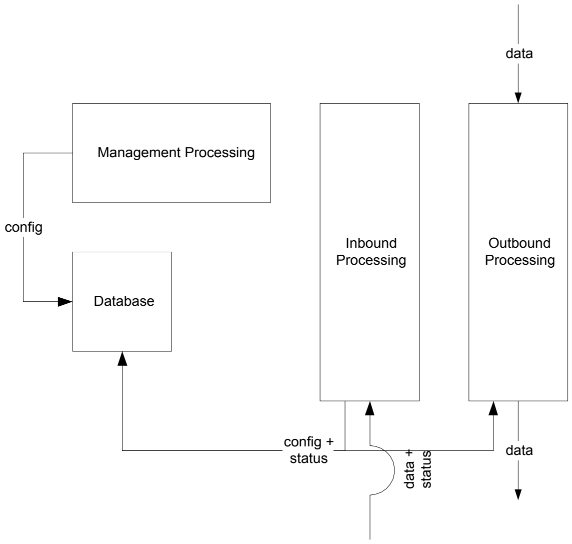


Figure 1-2.21. Transport-Adaption Layer: Architecture

2.3.5.3.2. Data Flow Diagrams and Process Specifications

Detailed design information is provided in Appendix 1.

2.3.5.4. Additional notes

The following issues were addressed in the design.

- Possible additional control via Management -- The construction of management processing allows for new functionality to be added with minimal disruption to the existing design.
- Possible additional interpretation of Transport Messages -- The inclusion of stub modules allows for the insertion of processing for the Transport Layer messages that are currently not examined.
- Use of type switching for Messages and IEs -- The switches that classify Transport Layer Messages and Information Elements are such that they can be used on hierarchically typed data structures, an inherent ability for BONeS.
- Consistent architectural split -- The same architectural split has been employed as with most other modules; in terms of input, output and management delineation.
- Use of primitive accessors -- At all times, accessors are used to manipulate data structures.
- Placement of simulation Probes -- A number of stubs have been put into place to facilitate points at which Probes can be added for the purposes of collecting data during simulations.

2.3.6. Routing-Module

2.3.6.1. Overview

The Routing-Module is directly modelled from a real world entity. In the real world, routing is performed at the Network Layer. This involves an entity that receives messages from a set of Network Layers, and then delivers them back to the same set of Network Layers. However, the specific source and destination Network Layer for a specific message are not necessarily the same; the destination is generally decided by some kind of routing strategy.

This entity, a routing engine, can decide a destination based upon several sources of information. Generally, a routing table is specified indicating which Interface--Network Layer--is able to deliver the message to its ultimate destination--determined by an Address in the message: it may also contain other information to be used. The routing table may be statically defined, or subject to automated periodic updates (e.g. RIP, OSPF). However, the generally represent a medium to long-term determinant in the routing algorithm. More short-term information may be used, such as the state and load of a particular Interface.

Essential, the routing engine is a Layer 3 bridge. In an ideal environment, there is no distinction between a Network Layer in communication with a Transport Layer, or a Routing Module (or any other Layer, for that matter).

The model of the Routing-Module for our environment retains the key features just mentioned. It is connected to a number of Network Layers, and receives both Data and Status messages from these Layers. Data messages are routed via a conceptual routing engine and sent back to another Network Layer; whereas Status messages are used to update the known status of the Layer. The routing engine uses a table of routing entries, along with this status information, to determine which Network Layer should be the recipient of the message being routed--i.e. the routing algorithm.

Our model of a routing algorithm is inline with contemporary methods. The routing table entries define an Address, an Interface and a Cost. Each entry is only applicable to messages with a destination address corresponding to Address. As such, there may be a number of interfaces available for any given Address; hence some form of discrimination is required--noting that if a given Network Layer is unavailable, then entries corresponding to its Interface are ignored. The Cost is used with the current Load of the Interface and a scaling factor to compute a weight. The entry with the lowest weight, and generally the lowest queue size, is selected as the applicable routing entry--and therefore a provider of the Interface to which the messages is then directed to. The routing algorithm can be specified as follows:

1. Locate all *Routes* in the *Routing Table* with an *Address* equal to the *Message's Destination Address* and with an *Interface* that has an *Active Availability Status*.
2. For each *Route* that was located, select the one with the *Minimum Cost*; using the algorithm: COMPUTED COST := COST + ((INTERFACE LOAD) * BETA).

With the following qualifications:

- *Routes* are entries in the *Routing Table* that contain an *Address*, *Interface* and *Cost*.
- An *Address* is the address of a specific *End System* as contained in all *Messages* being routed.
- An *Interface* denotes a specific local *Network Layer*.
- The *Cost* is a specifically assigned weighing factor.
- The *Availability Status* is a list indicating wether a specific *Interface* is *Active* or *Inactive*.
- The *Interface Load* represents the current *Load* of an *Interface* as stored in the list.

The routing algorithm is simplistic and subject to caveats, yet it serves the requirements neatly. One specific requirement was for routing to be dynamic as a result of local congestion, which manifests itself by way of queue loading values. Therefore, with correct configuration, the destination for a message with a given address can be selected from a set of routing entries in such a way that it does alternate in output interface.

If no route exists for a given message, then it is discarded--there is no other option. Most real world routers have a default route, which is the last result if no others can be located. Additionally, the routing module is the place in which hop count checking occurs. Each message has a Hop Count that decrements as the message passes through intermediate systems. When the Hop Count reaches zero, the message is discarded.

Configuration of the Routing-Module is only concerned with routing table entries. The purpose here is to allow for these entries to be dynamically altered, so that specific situations can be contrived. The scaling factor used in the routing algorithm is fixed, and the interface status information is subject to update from Network Layer originated status messages.

The Routing-Module has simple operation. Fundamentally, it is a switch so there is a central point at which switching occurs. The surrounding framework is concerned with mapping to and from the Network Layers in an abstract and extensible manner, along with providing database functionality for the Network Layer state and Routing-Module configuration information.

2.3.6.2. External Interface

The external interface defines this entity as seen by other entities in the architecture and provides a concrete position from which to design the internal construction. The treatment of the external interface requires delineation of the other entities that are communicated with, the elements used for such communication, and the behavioural aspects of the communication.

2.3.6.2.1. Relationships

The Routing-Module is conversant with a number of entities; however these are logically represented as two classes of entities. The first class is a single Management entity that provides Management Messages to perform internal operations on the Routing-Module. The second type of entity is a set of Network Layers; these are viewed as a single parameterised Network Layer. Data Messages are received from a Network Layer, and transmitted to another Network Layer. Status information is also extracted from Network Layer messages and used.

Figure 1-2.22 illustrates the entities and their data relationships.

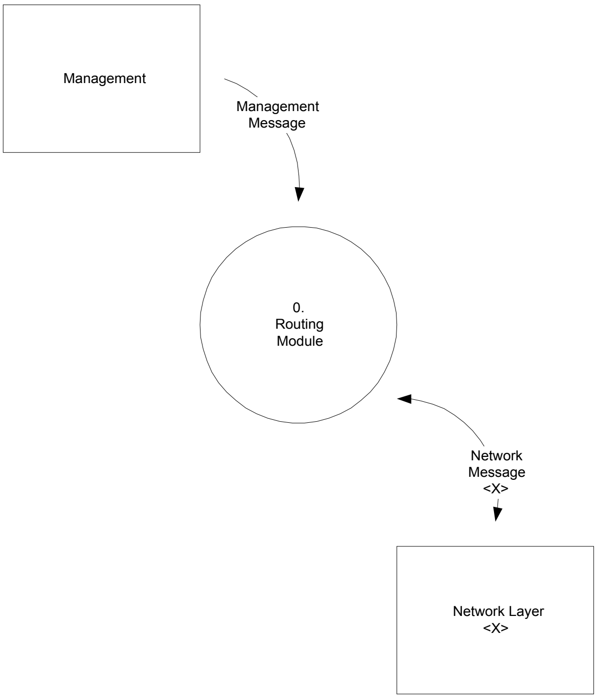


Figure 1-2.22. Routing-Module: Context Diagram

Table 1-2.16 details the roles and information communicated between entities.

Name	*Role*	*Communicated Information*
Network Layer <X>	Act to transfer messages across network paths	Network Layer Messages, both Status and Data information
Management	Modifies operation of Routing Module	Management Messages with IEs

Table 1-2.16. Routing-Module: Entity Relationships

Table 1-2.17 outlines the data.

Name	*Content*	*Purpose*
Network Connect Indication Message	Null	Indicates that specific Network Layer is able to send messages
Network Disconnect Indication Message	Null	Indicates that specific Network Layer is not able to send messages
Network Status Indication Message	IE	Indicates status information about specific Network Layer
Network Data Request Message	Address, Content, Length	Request transfer of message by Network Layer to given address
Network Data Indication Message	Address, Content, Length	Indicate reception of message by Network Layer destined for Address
Management Set Indication Message	IE	Provides IEs that modify Routing Module operation
Routing-Module Routing Entry IE	Address, Interface, Cost	Specify routing entry for use in routing arrived messages

Table 1-2.17. Routing-Module: Data Relationships

Any other data that arrives from external entities to the Routing-Module is ignored.

2.3.6.2.2. Parameters

Apart from the mandatory Address, there are no externally visible parameters for the Routing-Module. However, the Routing Table Entries are internal parameters that are directly modifiable by Management, therefore they are considered an externally visible parameter--just indirectly accessed. The parameters are shown in Table 1-2.18.

Name	*Purpose*	*Values*	*Default*	*Example*
Address	Allow messages to be directed to this module	Integer	0	10
Routing-Table Entry	Contain tuple of Address, Interface and Cost for directing messages	Set: (Integer, Integer, Real)	Null	(10, 1, 0.5)

Table 1-2.18. Routing-Module: Parameters

2.3.6.2.3. Behaviour

- *Network Data Indication Message* Input

This Indication arrives from the Network Layer. It is first processed by a Network Layer Interface that passes it to a central Routing Processor. This Routing Processor will use the *Address* of the message, the currently defined *Routing Table*, the known state of a particular interface, and the known load on a particular interface to compute a new destination Network Layer. The Indication is then passed to that Network Layer as a Request.

The Indication may be discarded if it is found that it has passed through too many Hops (a Hop Count is specified in the Indication and is used to ensure that messages do not loop indefinitely in the network). It may also be discarded if no Route is found.

- *Network Connect Indication Message* Input

When this message is received, it is a notification that the Network Layer is active. The respective Network Layer Interface updates a database entry to indicate that the Layer is active and able to have messages directed towards it.

- *Network Disconnect Indication Message* Input

When this message is received, it is a notification that the Network Layer is inactive. The respective Network Layer Interface updates a database entry to indicate that the Layer is inactive and therefore cannot have messages directed towards it.

- *Network Status Indication Message* Input

This message arrives from the Network Layer. It contains an indication of the current load state of the queues at the Network Layer normalised to be a real number between 0 and 1. The respective Network Layer Interface updates a database entry with this load which is used in route computation to aid in the selection of an route.

- *Management Set Indication Message* Input

The message is first verified to ensure that its Destination Address corresponds to the Routing-Module's *Address*. If it is not destined for this *Address*, then it is discarded. If accepted, the Information Element in the message is extracted and processed according to its type:

  - *Routing-Module Routing Entry IE* -- The contents of this IE are a tuple of values to define a *Routing Entry*. This tuple consists of the *Address* that the route is for, the *Interface* to be used for the route, and a *Cost* factor associated with the route.

- *Network Data Request Message* Output

This output occurs as a result of the routing of an input. The Network Layer is expected to deliver this message to the *Address* specified in the message.

2.3.6.2.4. Data Accessors

The only Routing-Module specific data is the *Routing-Module Routing Entry IE*, which is generated by Management and processed by the Routing-Module.

Name
Construct IE Routing-Module Route Entry Extract IE Routing-Module Route Entry

2.3.6.2.5. Dependencies

Due to the use of the Management and Network Layer entities are required by the Routing-Module.

Name
Extract Message Management Set Indication
Convert Message Network Data Indication To Request
Construct Message Network Data Request
Construct Message Application Data
Extract Message Network Connect Indication
Extract Message Network Disconnect Indication
Extract Message Network Data Indication
Insert Message Network Data Indication
Extract Message Network Status Indication

2.3.6.2.6. Initialisation

Each of the Network Layers is presumed to be inactive. Also, the *Routing Table* is empty, and the load of each Network Layer is presumed zero. The *Address* of the Routing-Module must be defined if Management Messages are to be received.

2.3.6.3. Internal Design

2.3.6.3.1. Approach

Internally, the Routing-Module consists of several core processing blocks, these are:

- *Management Processing* to process messages that arrive from Management, which currently are only defined to update the routing table.
- *Routing Engine* to route Network Layer Data Messages by accepting an input *Network Data Indication Message* and routing according to Routing Tables and other conditions (such as the state and load of a particular Network Layer).
- *Network Interface* to interface to each particular Network Layer connected to the Routing Module. Messages from each Network Layer are used to update network state and load information, along with being passed into, or accepted from, the *Routing Engine*.

The architecture is specifically designed so that *Network Interfaces* are aggregated in a manner that allows for easy expansion. It was also deemed important to separate the management entity and to provide a central abstract switching point that is not limited by any external components. These functional groupings are shown in the architectural diagram in Figure 1-2.23.

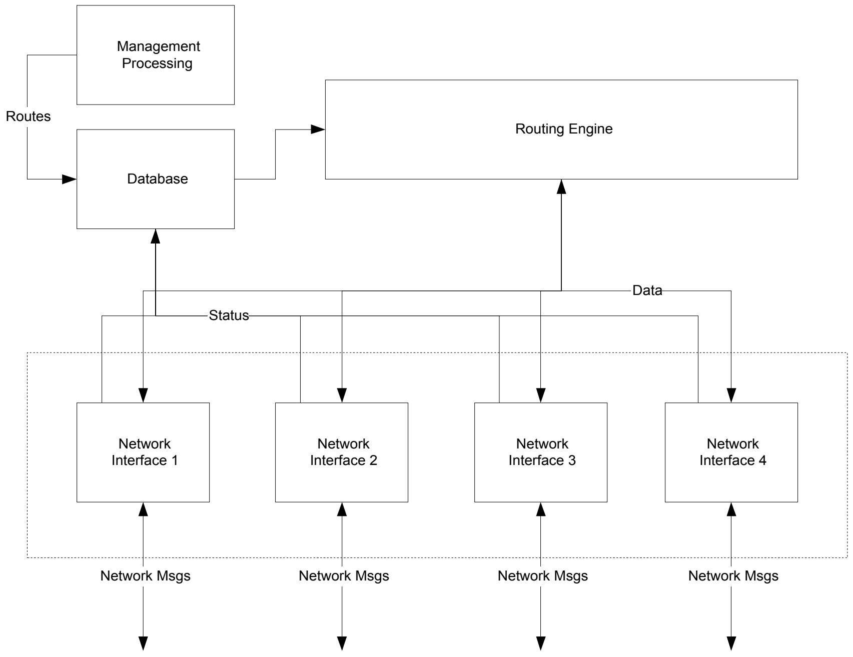


Figure 1-2.23. Routing-Module: Architecture

2.3.6.3.2. Data Flow Diagrams and Process Specifications

Detailed design information is provided in Appendix 1.

2.3.6.4. Additional notes

The following issues were addressed in the design.

- Possible additional control via Management -- The construction of management processing allows for new functionality to be added with minimal disruption to the existing design.
- Possible additional interpretation of Network Messages -- The inclusion of stub modules allows for the insertion of processing for the Network Layer messages that are currently not examined.
- Use of type switching for Messages and IEs -- The switches that classify Network Layer Messages and Information Elements are such that they can be used on hierarchically typed data structures, an inherent ability for BONeS.
- Use of primitive accessors -- At all times, accessors are used to manipulate data structures.
- Placement of simulation Probes -- A number of stubs have been put into place to facilitate points at which Probes can be added for the purposes of collecting data during simulations.
- Additional Network Layers -- The Network Layers are parameterised in such a way that it is trivial to insert new Network Layers by merely duplicating Network Interfaces.

- Modifications to Routing Algorithm -- The Compute Next Hop and Compute Cost processes in the Routing Module have been abstracted so that it is possible to modify them without any surrounding alteration.

2.3.7. Generator

2.3.7.1. Overview

The Generator models a source of traffic in the real world. An example of such an entity in the real world is an application; which is exactly the type of entity we desired to have modelled from the real world for the purpose of simulating.

The role of the Generator is to provide abstract elements of data (represented as a length of data---the content is irrelevant) at periodic intervals. The behaviour of this generation is described by its temporal and spatial characteristics---the time periods between data output, and the length at each output. This behaviour can be configured.

The configuration occurs by way of the Management entity. It is able to construct a request that describes a specific behavioural pattern that is to be used for the generation of output. There are two classes of patterns: real world samples and statistical profiles. The first consists of measured FTP and Telnet characteristics, and the second of Constant, Normal, Uniform, Exponential and Poisson variables. The generation of output in accordance to this behaviour will continue until a limit is reached (defined as a length of time, a total length, or a number of outputs) or a specific stop is requested---the second possible request from the Management entity.

Internally, the construction of the Generator is partitioned in such a way that it allows for ease of expansion. To produce the FTP and Telnet characteristics, the TCPLIB (TCPLIB) package is used. It can, when provoked, construct a variable corresponding to measured characteristics.

2.3.7.2. External Interface

The external interface defines this entity as seen by other entities in the architecture and provides a concrete position from which to design the internal construction. The treatment of the external interface requires delineation of the other entities that are communicated with, the elements used for such communication, and the behavioural aspects of the communication.

2.3.7.3. Relationships

The Generator concerns itself with only two external entities. The first of which is the Management entity, which sets up the Generator to operate with a specific behaviour. The second is the Lower Layer, which is the recipient of the data lengths that are generated.

Figure 1-2.24 illustrates the entities and their data relationships.

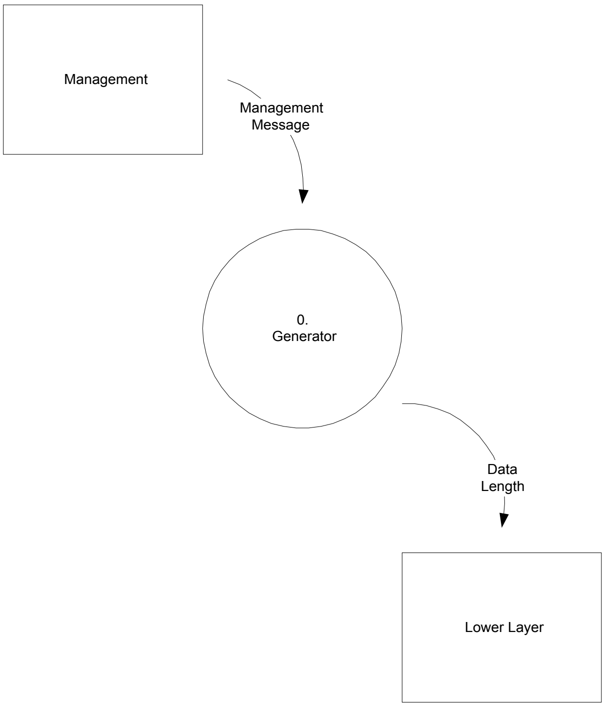


Figure 1-2.24. Generator: Context Diagram

Table 1-2.19 details the roles and information communicated between entities.

Name	*Role*	*Communicated Information*
Lower Layer	Accepts unit data elements	Data Length
Management	Modifies behaviour of the Generator	Management Messages with IEs

Table 1-2.19. Generator: Entity Relationships

Table 1-2.20 outlines the data.

Name	*Content*	*Purpose*
Management Set Indication Message	IE	Provides IEs used to setup and stop the Generator
Generator Stop IE	Null	Requests that output stop
Generator Setup Telnet IE	Filter Info	Requests initiation of Telnet profile generation
Generator Setup FTP IE	Filter Info	Requests initiation of FTP profile generation
Generator Setup Statistical IE	Filter Info, Stat Info	Requests initiation of Statistical profile generation

Table 1-2.20. Generator: Data Relationships

Any other data that arrives from the external entities to the Generator is ignored.

2.3.7.3.1. Parameters

Apart from the mandatory *Address*, there are no externally visible parameters for the Generator. The parameters are shown in Table 1-2.21.

Name	*Purpose*	*Values*	*Default*	*Example*
Address	To address messages to this module	Integer	0	10

Table 1-2.21. Generator: Parameters

2.3.7.3.2. Behaviour

- *Management Set Indication Message* Input

The message is first verified to ensure that its Destination Address corresponds to the Generator's *Address*. If it is not destined for this *Address*, then it is discarded. If accepted, the Information Element in the message is extracted, and processed according to its type:

- *Generator Setup Telnet IE* -- The Generator is configured to produce *Data Output* using samples of Telnet characteristics. This IE also contains Filter parameters.
- *Generator Setup FTP IE* -- The Generator is configured to produce *Data Output* using samples of FTP characteristics. This IE also contains Filter parameters.
- *Generator Setup Statistical IE* -- The Generator is configured to produce *Data Output* according to a defined statistical profile. There are two characteristics present, one is for the temporal behaviour (time between generation) and the other is for the spatial behaviour (the size of each item generated). The profiles can be Constant, Normally Distributed, Uniformly Distributed, Exponentially distributed or Poisson variables -- each have a specific set of parameters. This IE also contains Filter parameters.
- *Generator Stop IE* -- The Generator will stop producing output.

- *Data Length* Output

This output occurs as a result of the Generator constructing it using its current configuration.

2.3.7.3.3. Data Accessors

For constructing and deconstructing data that flows into and out of the Generator, a number of accessors are designed to encapsulate the direct access to the data.

Name
Construct IE Generator Setup Telnet
Construct IE Generator Setup FTP
Construct IE Generator Setup Statistical
Construct IE Generator Stop
Extract IE Generator Setup Telnet
Extract IE Generator Setup FTP
Extract IE Generator Setup Statistical
Extract IE Generator Stop

2.3.7.3.4. Dependencies

The Generator requires external modules.

Name
Extract Message Management Set Indication
TCP Library
Statistical

2.3.7.3.5. Initialisation

The *Address* of the Generator must be defined if any Management Messages are to be received.

2.3.7.4. Internal Design

2.3.7.4.1. Approach

The internal architecture is partitioned into the following main processing blocks:

- *Setup Generator* as an entity to accept and process the Setup IEs arriving from Management, and thence to extract the profile and filter parameters and use these to start and operate an instance of generator activity.
- *Stop Generator* as an entity to accept and process the Stop IEs arriving from Management, and thence to actually stop any current generator activity that may be in operation.

The partitioning occurred largely due to the internal partitioning requirements of the Setup Generator where all specific instances of processes that can generate data are subject to filtering. Hence, it was appropriate not to "mix" this level with that above.

2.3.7.4.2. Data Flow Diagrams and Process Specifications

Detailed design information is provided in Appendix 1.

2.3.7.5. Additional notes

The following issues were addressed in the design.

- Possible additional control via. Management -- The Management processing is such that additional IE's could be processed with a minimum of disturbance to the current design.
- Possible additional types of generation -- The parsing of the Setup IE is such that additional classifications of Setup IE could be processed and thus there could be additional types of parameters that could be generated. It was envisaged that HTTP characteristics could be added here, but none were available to date.
- Possible expansion of Statistical Types -- The use of the abstract Statistical module allows for internal changes to be carried out such that additional statistical types could be added without having to modify the Generator at all.
- Non-Network aware design -- The design is such that the Generator has no knowledge of other Modules other than its Management controller. The purpose of this is to allow the Generator to be re-used in various ways in the environment.

2.3.7.6. The TCB Library

The TCP Library (TCPLIB) is a software program developed as a result of collecting Wide-Area Traffic statistics for various protocols that use TCP as a transmission agent. The purpose of the library is to provide actual samples, as collected, of specific characteristics of the measured protocols. The importance is that it directly uses known samples, as opposed to modelling these via a statistical mechanism.

The library provides abstract function hooks that allow for the retrieval of a random variable based on the known samples. For each protocol (e.g. Telnet (RFC854, 1983), FTP (RFC959, 1985), SMTP (RFC821, 1982) and so on) there are a number of characteristics that have been collected. In the case of Telnet, for example, there is a packet inter-arrival time, a packet size, and a conversation length.

Software can make use of these simply by calling the function hooks, and the implementation must make use of the 'C' interface to BONeS for this ability. The interface is simple, as the there is no input, and only an "integer" or "real" output.

2.3.8. Management

2.3.8.1. Overview

Management is a module specifically for the purpose of managing other modules in the simulation environment. It does not model a specific entity in the real world, although many OSI based architectures do have the notion of just such a management entity, our purpose here is slightly different. This module is constructed to serve the requirements of dynamic control during the execution of a simulation.

The functionality of Management is to construct and transmit Information Elements (IEs) at a specific time to a specific destination. The time, destination and IE contents are read from a file. Management carries out a process of reading each item individually to build up the required elements in the IE. Error checking is performed to cope with short file reads and erroneous content.

Internally, the construction was specifically done in such a way to allow for the expansion of Management both in terms of the modules that can be managed, and the IEs that can be sent to these modules.

2.3.8.2. External Interface

The external interface defines this entity as seen by other entities in the architecture and provides a concrete position from which to design the internal construction. The treatment of the external interface requires delineation of the other entities that are communicated with, the elements used for such communication, and the behavioural aspects of the communication.

2.3.8.2.1. Relationships

Management communicates with two classes of external entities. The first class, which has only one entry, is an initialisation subsystem which is able to kick start Management as the first thing in the execution of a simulation. This ensures that Management starts reading the file. The second class consists of every other module in the simulation environment. In the case of the latter, a module is qualified by a unique Address: Management will send a Message to a module with a specified Address.

Figure 1-2.25 illustrates the entities and their data relationships.

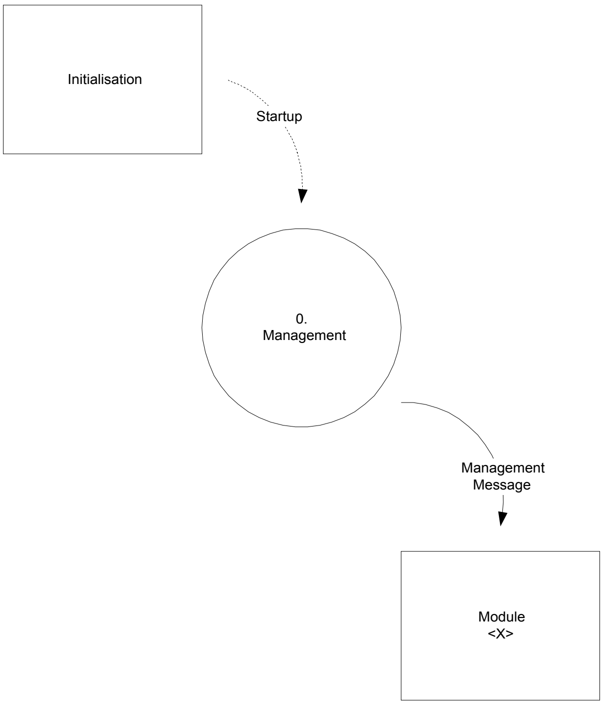


Figure 1-2.25. Management: Context Diagram

Table 1-2.22 details the roles and information communicated between the entities.

Name	*Role*	*Communicated Information*
Module <X>	Accepts Management Messages that change Module's behaviour	Management Message, with IEs
Initialisation	Starts up the Management module	Startup, initial control kick

Table 1-2.22. Management: Entity Relationships

Table 1-2.23 outlines the data.

Name	*Content*	*Purpose*
Startup	Null	Start up processing
Management Set Indication Message	Address, IE	Indicate IE that Address should process
Transport-Adaption Connect IE	Address	Tell Transport-Adaption Layer to connect session to address
Transport-Adaption Disconnect IE	Null	Tell Transport-Adaption Layer to disconnect session to address
Network-Adaption Address List IE	List of Address	Tell Network-Adaption Layer list of addresses to be used for messages
Datalink State IE	Boolean	Tell Datalink Layer to be active or inactive
Transport Setup IE	ISN	Tell Transport Layer the ISN to use for setup sessions
Routing-Module Routing Entry IE	Address, Interface, Cost	Tell Routing-Module a new routing entry to use
Generator Setup Telnet IE	Filter Info	Tell Generator to start generating Telnet traffic
Generator Setup FTP IE	Filter Info	Tell Generator to start generating FTP traffic
Generator Setup Statistical IE	Filter Info, Stat Info	Tell Generator to start generating statistical traffic
Generator Stop IE	Null	Tell Generator to stop generating traffic

Table 1-2.23. Management: Data Relationships

Any other data that arrives from external entities to Management is ignored.

2.3.8.2.2. Parameters

There is only one parameter that Management requires and this is shown in Table 1-2.24.

Name	*Purpose*	*Values*	*Default*	*Example*
Filename	Provide the file from which Management commands are read.	Complete path and filename.	Null	/tmp/sim.txt

Table 1-2.24. Management: Parameters

2.3.8.2.3. Behaviour

- *Startup* Input

This notification arrives from an Initialisation Subsystem. It is used to open the Management file, and to start reading entries from the file. If the file cannot be opened, then errors will be reported.

- *Management Set Indication Message* Output

This message is generated as the result of an entry having been processed from the Management file. The message contains a single Information Element (IE) and is directed towards a specific Address. The receiving module is expected to process this message.

2.3.8.2.4. Data Accessors

There is only one message that is particular to the Management module.

Name
Construct Message Management Set Indication
Extract Message Management Set Indication

2.3.8.2.5. Dependencies

Due to the use of external Information Elements, there are a number of external data accessors that are required.

Name
Construct IE Transport-Adaption Connect
Construct IE Transport-Adaption Disconnect
Construct IE Network-Adaption Address List
Construct IE Datalink State
Construct IE Transport Setup
Construct IE Routing-Module Routing Entry
Construct IE Generator Setup Telnet
Construct IE Generator Setup FTP
Construct IE Generator Setup Statistical
Construct IE Generator Stop

2.3.8.2.6. Initialisation

At initialisation, an activation is given to Management so that it may begin processing the management file.

2.3.8.3. Internal Design

2.3.8.3.1. Approach

The internal construction of the Management Module consisted of partitioning processing procedurally. The major steps in the process are:

1. At startup, the file is opened.
2. The next entry is located, and processing is suspended until the time indicated in the entry.
3. The destination address and module type is read from the entry.
4. The module type is used to parse the rest of the entry and construct an Information Element (IE).

5. A Management Message is constructed and sent off to the destination with the IE.
6. Process goes back to locating another entry (Step 2).
7. Any errors that occur are reported and logged.

The segregation of the steps above was also done to ensure that processing was partitioned into the following components (as mapped into the steps above) which are deemed to be significant in terms of extensibility.

1. Initialisation
2. Wait
3. Get Address
4. Get IE (qualified on Address)
5. Send IE to Address
6. Iterate
7. Indicate Errors

The 4th step is where most extensions will occur, as it is the place that IEs are parsed according to the type of module that they are destined for.

2.3.8.3.2. Data Flow Diagrams and Process Specifications

Detailed design information is provided in Appendix 1.

2.3.8.4. Additional notes

The following issues were addressed in the design.

- Possible additional Modules -- If there were new modules added to the environment, the modifications to insert processing for the new module would be trivial to the current design.
- Possible additional IEs for Modules -- The processing of more IEs within each module are similarly trivial modification, again requiring an additional process and a change to the switching mechanism.
- Use of primitive accessors -- At all times, primitive accessors have been used to encapsulate the accessing of data structures, thus allowing for underlying data structure change without modification to existing references.

2.4. Miscellaneous Modules

The miscellaneous modules play minor, supporting, roles in the architecture.

2.4.1. Statistical Parameter

This module encapsulates the ability to generate a random number of specific type. It includes a data structure within which the parameters for the types are hidden, and abstract modules to take the data structure and produce an output. The following types of parameters may be created.

- **Constant** -- Create constant variable of defined value.
- **Uniform** -- Create a uniformly distributed variable between given Minimum and Maximum values.
- **Normal** -- Create a normally distributed variable using given Mean and Variance values.
- **Exponential** -- Create an exponentially distributed variable of given mean value.
- **Poisson** -- Create a poisson distributed variable of given lambda value.

2.4.1.1. Data Structure

The data structure is a composite union:

```
Statistical Info :=                 Statistical Info Constant |
                                    Statistical Info Uniform |
                                    Statistical Info Normal |
                                    Statistical Info Exponential |
                                    Statistical Info Poisson.

Statistical Info Constant :=        Value : REAL.

Statistical Info Uniform :=         Minimum : REAL,
                                    Maximum : REAL.

Statistical Info Normal :=          Mean : REAL,
                                    Variance : REAL.

Statistical Info Exponential :=     Mean : REAL.

Statistical Info Poisson :=         Lambda : REAL.
```

2.4.1.2. Functions

The externally visible functions, as shown in Table 1-2.25, are provided, with their PSPECs outline in Appendix 1.

Name and Prototype	*Description*
Get_Statistical_Info (Statistical Info) : Real	Given the abstract statistical information structure, this function will generate an appropriate value for the defined type of parameter to be created.
Create_Stat_Constant (Value) : Statistical Info	Create an abstract statistical information structure for the purposes of generating constant values.
Create_Stat_Uniform (Lower, Upper) : Statistical Info	Create an abstract statistical information structure for the purposes of generating uniformly distributed values.
Create_Stat_Normal (Mean, Variance) : Statistical Info	Create an abstract statistical information structure for the purposes of generating normally distributed values.
Create_Stat_Exponential (Mean) : Statistical Info	Create an abstract statistical information structure for the purposes of generating exponentially distributed values.
Create_Stat_Poisson (Lambda) : Statistical Info	Create an abstract statistical information structure for the purposes of generating poisson distributed values.

Table 1-2.25. Statistical Parameter: Functions

The concerns in constructing this were:

- Ease of inserting new statistical parameter; by just inserting another field into the composite union and providing another function to compute the value.
- Ability to use BONeS type determination mechanisms to classify the composite union -- this parallels polymorphic behaviour with virtual tables.
- Ability to use BONeS data hierarchy and inheritance to define a composite union.

2.4.2. Probe - Transport Layer TCP

2.4.2.1. Overview

During the execution of a simulation, Probes are used to collect items of data (datum) for the eventual purpose of post-processing, interpretation and analysis. BONeS defines general Probes to capture datum from primitive data structures, however in the this design the TCP is implemented in the 'C' language. Hence, the internal details of the TCP are outside the reach of usual BONeS Probes. For this reason, a specific TCP Probe is constructed.

The TCP Probe collects data that resides within a TCP, according to the particular TCP Number it receives through an input port. A parameter to the TCP Probe identifies the particular datum to gather, and whether or not updates are restricted to being differential only. There are a number of Data Types that can be examined, most of which are direct copies from the TCP TCB, however this need not be the case. The particular Data Types currently defined are a result of the requirements for simulation design.

The output of the Probe is a REAL type, which BONeS stores and allows to be post-processed. For example, the value of the TCP Congestion Window could be output whenever it changes.

2.4.2.2. External Interface

The external interface is simple in nature. The input consists of an INTEGER, which corresponds to the TCP Number. The output consists of a REAL. In addition, there are two parameters that define the internal operation of the Probe.

2.4.2.2.1. Parameters

There are two parameters of importance; these are set when the Probe is placed into a Simulation. These parameters are provided in Table 1-2.26.

Name	*Purpose*	*Values*	*Default*	*Example*
Data Type	Indicate what item of data the Probe is to output.	Defined in other table.	Null	Send Window
Changes Only	Indicate whether we want Changes Only, or the data itself on each update.	Boolean True/False	True	True

Table 1-2.26. Transport Layer TCP Probe: Parameters

2.4.2.2.2. Behaviour

When the TCB Number is received on the Probe's input, the TCB corresponding to the TCB Number is interrogated and used to compute the particular datum required according to the *Data Type*. If this datum is equivalent to the previous datum, and *Changes Only* is *True*, then no output occurs. Otherwise, for *Changes Only* of *False*, output does occur.

Table 1-2.27 illustrates the *Data Type*'s that are defined.

Data Type	*Description*
Congestion Window	Provides the Congestion Window size.
Slow Start Threshold	Provides the Slow Start Threshold.
Retransmission Event	Indicates that a Retransmission occurred.
Round Trip Time Average	Provides the RTT Average (as computed by TCP).
Round Trip Time Variance	Provides the RTT Variance (as computed by TCP).
Send Window	Provides the Send Window size.
Unacknowledged Data	Provides the amount of unacknowledged data.
Timer Expiry	Indicates that a Timer Expiry occurred.
Acknowledgement Received	Indicates the particular acknowledgement received.
Reassembly Queue Length	Indicates the size of the Reassembly Queue.
Kilobytes Retransmitted	Indicates the total Kilobytes retransmitted.
Kilobytes Sent	Indicates the total Kilobytes sent.

Table 1-2.27. Transport Layer TCP Probe: Data Types

2.4.2.3. Internal Design

Detailed design information is provided in Appendix 1.

2.4.2.4. Additional notes

- The TCB Number must be passed as an input, as BONeS will not allow exported memory parameters for Probes. Non-exported parameters are allowable, hence the ability to contain the *Data Type* and *Changed Only*.

2.4.3. Probe - Network Layer Queue

2.4.3.1. Overview

The Network Layer consists of a Queue implemented in the 'C' language. As this Queue is not accessible by BONeS Probes, a specific Queue Probe is constructed.

The Queue Probe collects data about a Queue according to the particular Queue Number it receives through an input port. A parameter to the Queue Probe identifies the particular datum to gather, and whether or not updates are restricted to being differential only. There are a number of Data Types that can be examined; those currently defined are a result of requirements in simulation design.

The output of the Probe is a REAL type, which BONeS stores and allows to be post-processed. For example, the value of the Queue Size could be output whenever it changes.

2.4.3.2. External Interface

The external interface is simple in nature. The input consists of an INTEGER, which corresponds to the Queue Number. The output consists of a REAL. In addition, there are three parameters that define the internal operation of the Probe.

2.4.3.2.1. Parameters

There are three parameters of importance; these are set when the Probe is placed into a Simulation. These parameters are provided in Table 1-2.28.

Name	*Purpose*	*Values*	*Default*	*Example*
Data Type	Indicate what item of data the Probe is to output.	Defined in other table.	Null	Send Window
Changes Only	Indicate whether we want Changes Only, or the data itself on each update.	Boolean True/False	True	True
Address	Used for Data Types that need to qualify operation to a particular Address.	Integer	0	10

Table 1-2.28. Network Layer Queue Probe: Parameters

2.4.3.2.2. Behaviour

When the Queue Number is received on the Probe's input, the Queue corresponding to the Queue Number is interrogated and used to compute the particular datum required according to the *Data Type*. If this datum is equivalent to the previous datum, and *Changes Only* is *True*, then no output occurs. Otherwise, for *Changes Only* of *False*, output does occur.

Table 1-2.29 illustrates the *Data Type*'s that are defined.

Data Type	*Description*
Queue Size	Provides the current Queue size.
Queue Src Address Count	Provides the number of items with Src Address <X> in the Queue.
Queue Dest Address Count	Provides the number of items with Dst Address <X> in the Queue.

Table 1-2.29. Network Layer Queue Probe: Data Types

2.4.3.3. Internal Design

Detailed design information is provided in Appendix 1.

2.4.3.4. Additional notes

- The Queue Number must be passed as an input, as BONeS will not allow exported memory parameters for Probes. Non-exported parameters are allowable, hence the ability to contain the *Data Type*, *Changed Only* and *Address*.

2.5. Components

The components as specified in the architecture are constructed from the designed modules. This is a trivial exercise carried out by performing the aggregation as indicated in the architecture. These aggregations are more concretely specified here.

2.5.1. Simulation

At the top level, a simulation consists of Network Components and a Management Component. There can only be one Management Component, but any number of Network Components.

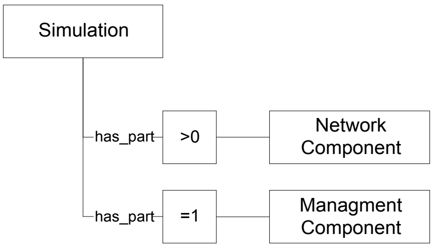


Figure 1-2.26. Simulation Component

The Management Component is the Management module in itself, which has a Filename from which commands are read. It is connected to other modules via a management port.

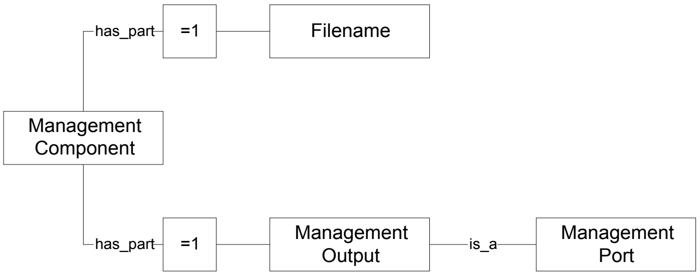


Figure 1-2.27. Simulation - Management Component

2.5.2. Network Components

All Network Components have a single Address, so that they are reachable from each other and from Management. They also have a management port for communication with Management. The Network Component is an End System, Intermediate System or Communications Link.

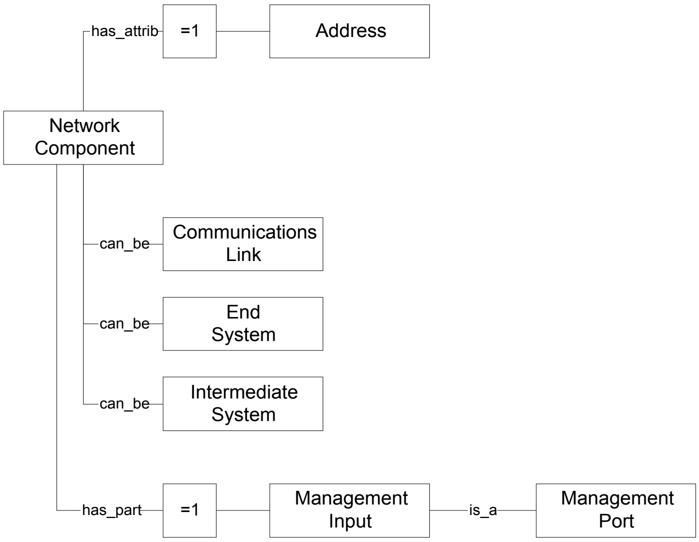


Figure 1-2.28. Simulation - Network Component

The Communications Link is merely a Datalink Layer.

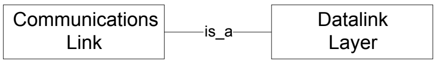


Figure 1-2.29. Network - Communications Link Component

The End System is a Host or is a Traffic Generator, both of which have a Network Layer.

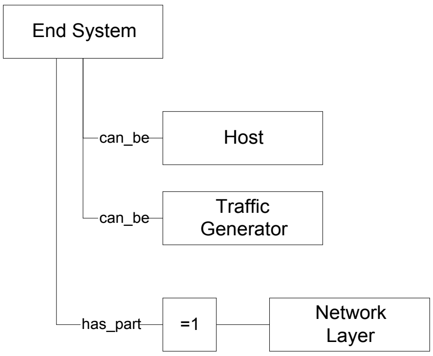


Figure 1-2.30. Network - End System Component

The Host contains a Generator, Transport-Adaption Layer and Transport Layer as well.

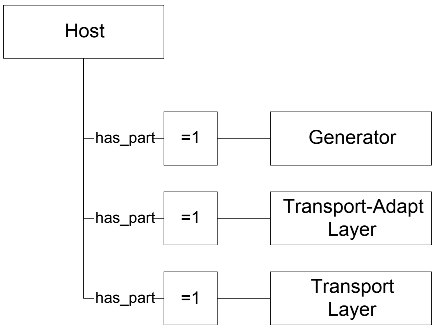


Figure 1-2.31. Network - Host Component

The Traffic Generator only contains a Generator and a Network-Adaption Layer, in addition to the Network Layer it inherits from above.

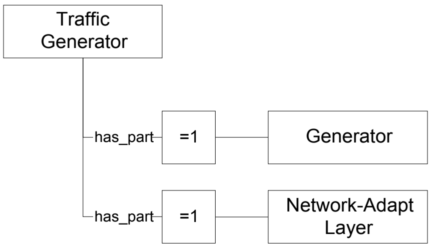


Figure 1-2.32. Network - Traffic Generator Component

The Intermediate System has a single Routing Module, but may have any number of Network Layers.

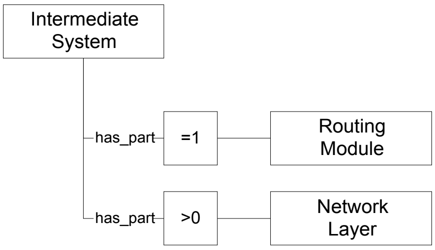


Figure 1-2.33. Network - Intermediate System Component

Each of these mention modules has specific input and output ports, these must be matched together when they are aggregated within a component. The following illustrates these constraints with respect to each module.

The Datalink Layer provides two Datalink Layer Ports.

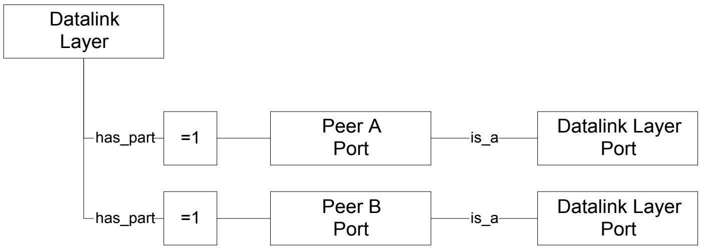


Figure 1-2.34. Module - Datalink Layer Component

The Network Layer provides a Network Layer Port, and uses a Datalink Layer Port.

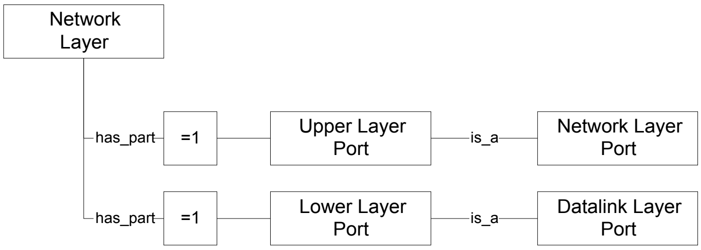


Figure 1-2.35. Module - Network Layer Component

The Transport Layer provides a Transport Layer Port, and uses a Network Layer Port.

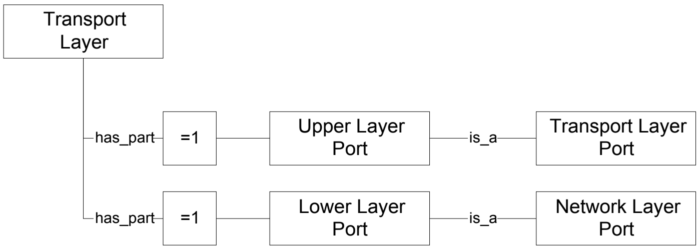


Figure 1-2.36. Module - Transport Layer Component

The Network-Adaption Layer provides a Data Port, and uses a Network Layer Port.

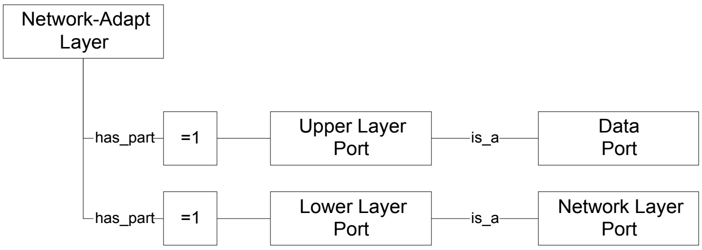


Figure 1-2.37. Module - Network-Adaption Layer Component

The Transport-Adaption Layer provides a Data Port, and uses a Transport Layer Port.

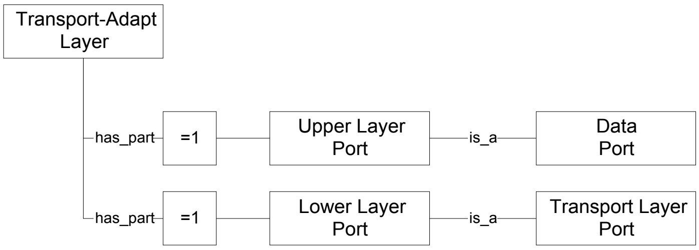


Figure 1-2.38. Module - Transport-Adaption Layer Component

The Generator provides a Data Port.

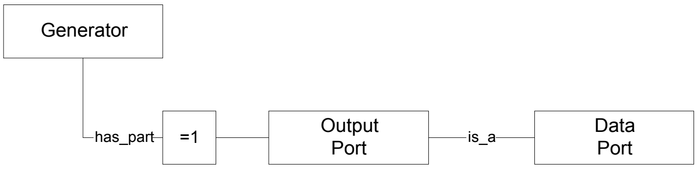


Figure 1-2.39. Module - Generator Component

The Routing-Module uses a number of Network Layer Ports.

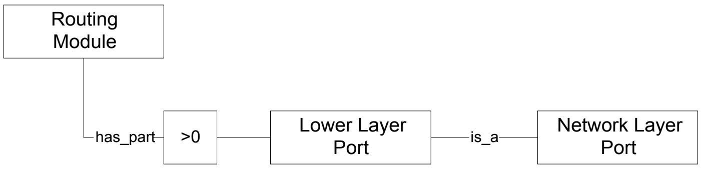


Figure 1-2.40. Module - Routing-Module Component

3. Implementation

The implementation was carried out by translating the designed Data Structures, Modules and Components from their logical representation into a BONeS representation of Data Structures and Modules. In this section, discussion is given on the strategies employed in this process, and the subsequent results from the process.

Due to the level of detail, only the top-level aspects are described in this section: the detailed lower levels are provided in Appendix 2. There are also notes provided for intended users of this implementation.

3.1. Strategies

In the implementation of the Modules and Data Structures -- particularly the Modules -- a consistent approach was taken on various aspects for various reasons. Principally, the implementation was mapped as directly from the design as possible; DFDs and PSPECs tended to map directly into individual BONeS modules -- both in structure and name -- and data stores into BONeS memory and parameters. The choice between using memory and parameters was based on an evaluation of whether the particular item was subject to change during execution of a simulation.

The layout of BONeS modules is such that processing is generally directed from left to right, or up and down depending on the overall nature of the BONeS module. This retains a consistency that hopefully results in a clearer picture for potential users. In addition, where possible, sections of a BONeS module that are more closely related than other sections are more closely grouped, for the purposes of retaining logical relationships.

Where BONeS primitives have been used, they are often renamed to better suit the context within which they are operating; generally this is strongly related to the arguments supplied to the BONeS primitive. For example, the BONeS Delay primitive used to model the propagation delay of a transmission line is renamed to Delay for Propagation Delay. This presents a much clearer picture of that primitive's role.

The design has also accounted for stub BONeS modules; the purpose of which is to provide points at which probes can be placed. This strategy is preserved in the implementation and allows for use of the module to monitor various data structures for whatever reason without having to alter the module. This is a trivial accommodation, but worthwhile.

Performance wise, there are two main strategies employed. When Data Structures fan out from a BONeS module, copies of that module are created. This can lead to unnecessary overhead when the Data Structures are large. Where possible, attempts have been made to prevent unnecessary fan out. The other strategy is the implementation of critical sections in the 'C' language. This occurs with the Transport TCP protocol and the Network Queue. It should have also occurred with the Routing-Module's route computation.

Finally, re-use and modularity are attempted with common sets of modules that perform a clear and distinction function, when that function is used in many places. An example of this is the construction of a Message Switch, which internally consists of a several modules and type checking mechanisms.

3.2. Architecture

The mapping from design to implementation retained the same basic architecture, however there were some minor alterations.

3.2.1. Module Organisation

In the implementation, the Modules are organised in the same manner as the design; i.e. ordering them in a hierarchy corresponding to the Data Flow Diagrams. BONeS allows for hierarchical structuring, but strangely enough does not allow the same Module names to appear regardless of their hierarchical position.

At the very top level, a "Wide-Area TCP" entry exists. In physical terms, the files reside on the EE RCC Network in the directory

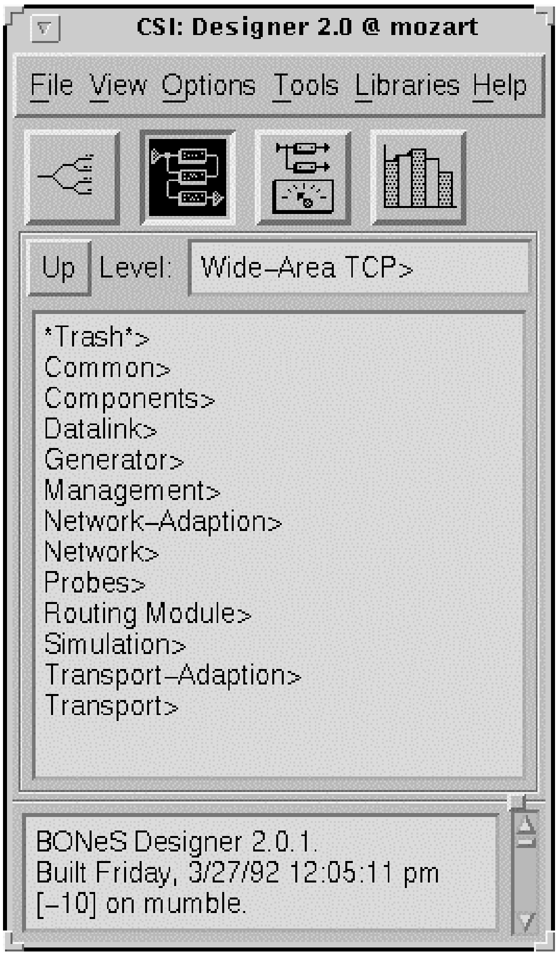
`~mgream/BONeS/Wide-Area_TCP`.

This directory also contains the stub 'C' language files used as primitives. For this entry, and all descended from it, to be present, the "Wide-Area TCP" library must be loaded.

Under the top level, each particular Module implemented has its own entry. Other entries exist, such as a Trash directory and a Simulations directory. The following table describes each entry.

- ***Trash*** -- Contains obsolete modules for backup purposes.
- **Common** -- Contains all of the Common Modules.
- **Components** -- Contains the Simulation Components.
- **Datalink** -- Contains the Datalink Layer Module and constituent modules.
- **Generator** -- Contains the Generator Module and constituent modules.
- **Management** -- Contains the Management Module and constituent modules.
- **Network-Adaption** -- Contains the Network-Adaption Layer Module and constituent modules.
- **Network** -- Contains the Network Layer Module and constituent modules.
- **Probes** -- Contains the Transport and Network Layer Probes.
- **Routing-Module** -- Contains the Routing-Module Module and constituent modules.
- **Simulation** -- Contains the actual Simulation Module and constituent modules.
- **Transport-Adaption** -- Contains the Transport-Adaption Layer Module and constituent modules.
- **Transport** -- Contains the Transport Layer Module and constituent modules.

A snapshot of the environment is shown in Figure 1-3.1, clearly illustrating these entries and the parent entry under which they are organised.


CSI: Designer 2.0 @ mozart
File View Options Tools Libraries Help
Up

Level: Wide-Area TCP>

Trash>
Common>
Components>
Datalink>
Generator>
Management>
Network-Adaption>
Network>
Probes>
Routing Module>
Simulation>
Transport-Adaption>
Transport>

BONeS Designer 2.0.1.
Built Friday, 3/27/92 12:05:11 pm
[-10] on mumble.

Figure 1-3.1. Top level Modules in BONeS

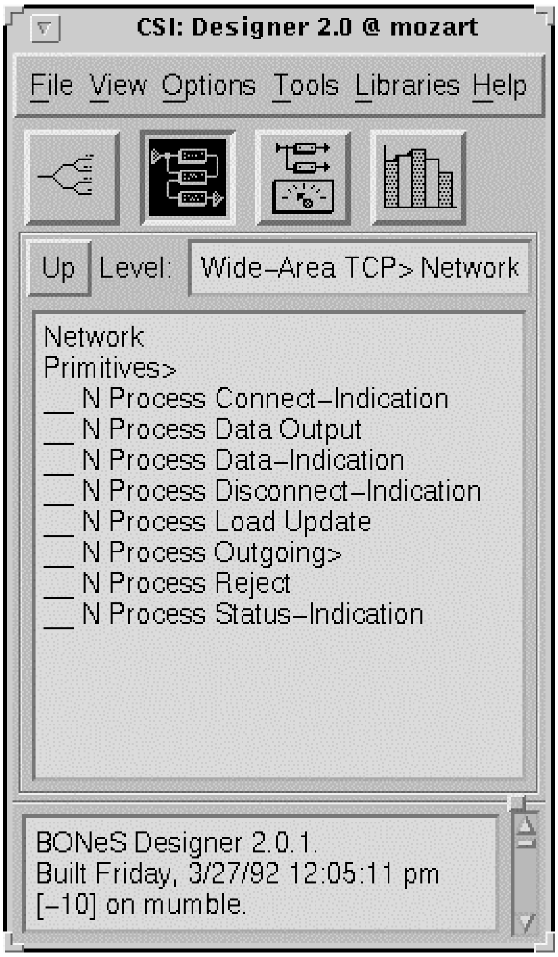
Within each entry, a number of conventions have been adopted.

Firstly, according to the design there are two aspects to each Module. There is the Module itself, and there are a number of Data Accessors used for the manipulation of Data Structures related to that Module. All Data Accessors are contained within a "Primitives" entry; there are no further levels within the "Primitives" entry.

Modules themselves are partitioned according to the Data Flow Diagrams and Process Specifications used in the design. The very top level of the DFD is contained as a single entry having the same name as a Module, e.g. "Network" under the "Network" Layer. For each level corresponding to a level in the DFD (including PSPECS) all bubbles at that level are contained within an entry having the same or similar name. Importantly, all entries are prefixed with a double underscore and letters corresponding to the entry for that level. For example, for a "Management" entry, all entries within it have "__ M" prefix. For a "Process Message" entry, all entries within it would have a "__ PM" prefix. The entries either correspond to actual BONeS Modules, or to another deeper level. At each new level, the convention is the same: the top entry has the same name as the entry from which it has descended, but with the prefix removed, and all bubbles at that level correspond to entries with prefixing in place.

It might be asked why this fuss occurred? Initially, the convention was simple: at each level, use the full name of the bubbles from the Data Flow Diagrams. But then it was discovered that BONeS does not allow names to clash, irrespective of where they are in the totality of all modules that it knows about at any given time. This absurdity would cause clashes for common names such as “Process Network Message”, “Management” and so forth, even when they were in different branches in the hierarchy. The solution was to adopt a consistent naming strategy, and using prefixes seemed an adequate solution. A prefix does not continue down to the next level for the reason that the length of the prefix would soon overshadow the name of the module.

The snapshot, shown in Figure 1-3.2, of the Network Layer illustrates these conventions. The “Primitives” entry exists, along with the “Network” entry corresponding to a BONeS module. All other entries, corresponding to bubbles in the DFD, are prefixed with “__ N” and only one entry “__ N Process Outgoing” leads to a lower level (which is apparent due to the trailing “>“).


CSI: Designer 2.0 @ mozart
File View Options Tools Libraries Help
Up
Level:

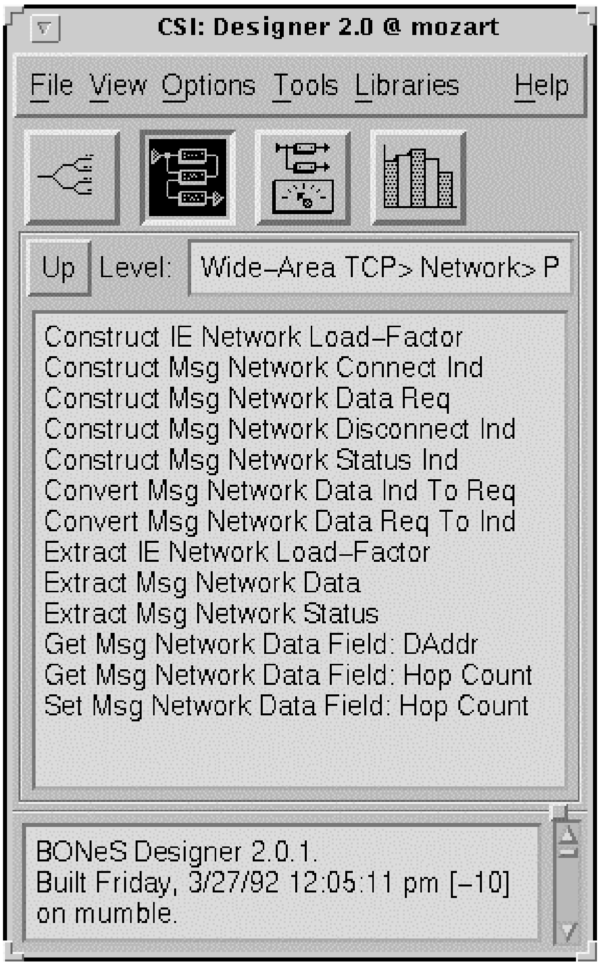
Wide-Area TCP> Network

Network
Primitives>
__ N Process Connect-Indication
__ N Process Data Output
__ N Process Data-Indication
__ N Process Disconnect-Indication
__ N Process Load Update
__ N Process Outgoing>
__ N Process Reject
__ N Process Status-Indication

BONeS Designer 2.0.1.
Built Friday, 3/27/92 12:05:11 pm
[-10] on mumble.

Figure 1-3.2. Network Layer Module in BONeS

Within the “Primitives” entry, all constructed Data Accessors for the Network Layer are present. This level is shown in Figure 1-3.3.



CSI: Designer 2.0 @ mozart
File View Options Tools Libraries Help
Up Level: Wide-Area TCP> Network> P

Construct IE Network Load-Factor
Construct Msg Network Connect Ind
Construct Msg Network Data Req
Construct Msg Network Disconnect Ind
Construct Msg Network Status Ind
Convert Msg Network Data Ind To Req
Convert Msg Network Data Req To Ind
Extract IE Network Load-Factor
Extract Msg Network Data
Extract Msg Network Status
Get Msg Network Data Field: DAddr
Get Msg Network Data Field: Hop Count
Set Msg Network Data Field: Hop Count

BONeS Designer 2.0.1.
Built Friday, 3/27/92 12:05:11 pm [-10]
on mumble.

Figure 1-3.2. Network Layer Data Accessors in BONeS

The purpose of this consistent and logically laid out structuring is to preserve understandability and traceability from the design to the implementation.

3.2.2. Data Structure Organisation

The Data Structures can be placed into three distinct categories.

The first category contains all the Information Elements that are used to transport items of information throughout the environment. As mentioned in the design, the Information Elements are constructed as BONeS COMPOSITE types, having a common “IE Primitive” root. From this, the tree branches first according to the particular Module that the Information Element corresponds to, and then according to the type of Information Element.

In BONeS, this inheritance allows for us to easily test the type of Information Element at any particular level. In this case, the “IE Primitive” does not contain any fields, so descendants do not gain any fields from it. The only case in this hierarchy where branches inherit significant information from their parents is with “IE Generator Setup Primitive” which contains basic Filter parameters. A snapshot of the Data Structure Hierarchy for Information Elements is shown in Figure 1-3.4.

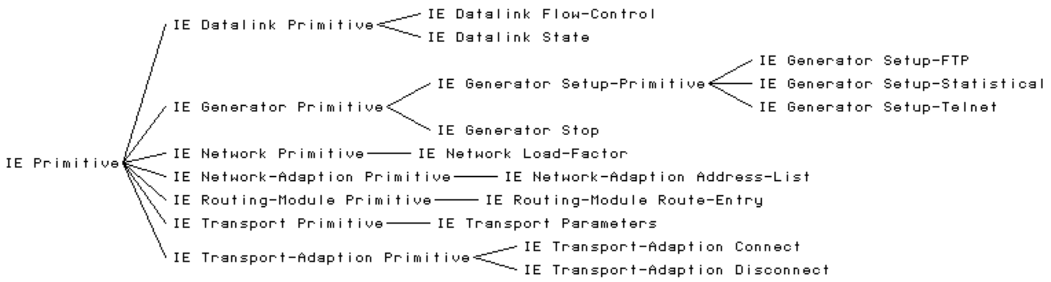

IE Primitive
IE Datalink Primitive
IE Datalink Flow-Control
IE Datalink State
IE Generator Primitive
IE Generator Setup-Primitive
IE Generator Setup-FTP
IE Generator Setup-Statistical
IE Generator Setup-Telnet
IE Generator Stop
IE Network Primitive
IE Network Load-Factor
IE Network-Adaption Primitive
IE Network-Adaption Address-List
IE Routing-Module Primitive
IE Routing-Module Route-Entry
IE Transport Primitive
IE Transport Parameters
IE Transport-Adaption Primitive
IE Transport-Adaption Connect
IE Transport-Adaption Disconnect

Figure 1-3.4. Data Structure Hierarchy for Information Elements in BONeS

The second category is that which contains all Messages used in the environment. The structuring here is much the same as that with Information Elements, in that they are constructed as BONeS COMPOSITE types, having a common "Msg Primitive" root. From this, the tree branches first according to the particular Module that the Message corresponds to, and then according to the type of the Message, and then finally according to the function qualification for the Message.

The "Msg Primitive" does contain fields to hold Length and Creation Date information. These fields are inherited by all descendants -- i.e. all Messages. It was considered that these two fields are basic elements in all Messages, therefore this level is appropriate to save unnecessary duplication. A snapshot of the Data Structure Hierarchy for Messages is shown in Figure 1-3.5.

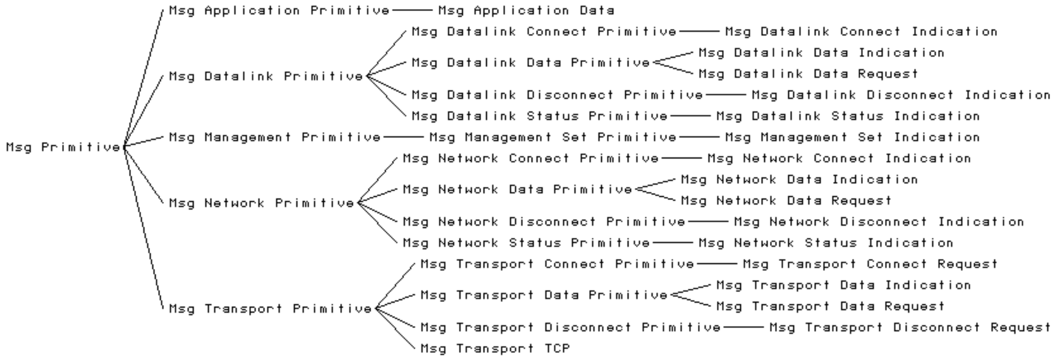

Msg Primitive
Msg Application Primitive
Msg Application Data
Msg Datalink Primitive
Msg Datalink Connect Primitive
Msg Datalink Connect Indication
Msg Datalink Data Primitive
Msg Datalink Data Indication
Msg Datalink Data Request
Msg Datalink Disconnect Primitive
Msg Datalink Disconnect Indication
Msg Datalink Status Primitive
Msg Datalink Status Indication
Msg Management Primitive
Msg Management Set Primitive
Msg Management Set Indication
Msg Network Primitive
Msg Network Connect Primitive
Msg Network Connect Indication
Msg Network Data Primitive
Msg Network Data Indication
Msg Network Data Request
Msg Network Disconnect Primitive
Msg Network Disconnect Indication
Msg Network Status Primitive
Msg Network Status Indication
Msg Transport Primitive
Msg Transport Connect Primitive
Msg Transport Connect Request
Msg Transport Data Primitive
Msg Transport Data Indication
Msg Transport Data Request
Msg Transport Disconnect Primitive
Msg Transport Disconnect Request
Msg Transport TCP

Figure 1-3.5. Data Structure Hierarchy for Messages in BONeS

It may be asked why there are many more Messages than are required, especially the proliferation of Primitives. The purpose for this structuring is that it allows for type testing and allows for much tighter constraints on ports that pass Messages. For example, with the Datalink Layer, the ports are constrained to all Messages of type "Msg Datalink Primitive" -- which covers that particular Message, and all descended from it. Not only is there better assurance for the operation of the system, but if any new Datalink Messages are added, then no changes need to be made to these ports. This same practice can occur down further in the tree, where an input port may be constrained to "Msg Data Primitive" only.

The third, and final, category encompasses other Data Structures used for ancillary and Miscellaneous purposes. This includes the Boolean Data Structure, which is constructed as a BONeS SET having the two values True and False, along with the Statistical Parameter which is constructed as a BONeS COMPOSITE having several descendants for more specialised types of Statistical Parameters (Constant, Exponential, ...). In this case, the inheritance is used for specification and information hiding purposes.

3.3. Modules

The categorisation of Modules as being Primary or Miscellaneous is continued from that used in the design. This section outlines the top-level implementation details; the detailed implementation aspects are provided in Appendix 2.

For each Module, any necessary details about the implementation are provided, which includes input and output Ports and types, Data Structures relevant to the Module and the top level BONeS Module for that Module. The detailed implementation aspects consist of further levels of modules, 'C' language source code and so forth.

3.3.1. Primary Modules

3.3.1.1. Datalink Layer

3.3.1.1.1. Overview

Implementation was straightforward, no major changes occurred. This Module was the first constructed, so acted as a test bed for subsequent Modules (which were considered, in the majority, to be more complex and involve more issues than this Module).

3.3.1.1.2. Ports

The design required interfaces for *Layer A*, *Layer B* and *Management*.

Management is implemented using the "portal" mechanism described as part of the Management implementation. *Layer A* and *Layer B* are implemented as two sets of input and output ports. The ports are listed in Table 1-3.1.

Name	*DS Type*	*Direction*	*Design*	*Notes*
Msg Request A	Msg Datalink Data Primitive	Input	Peer A (input)	The restriction is set to Data Messages only.
Msg Indication A	Msg Datalink Primitive	Output	Peer A (output)	
Msg Request B	Msg Datalink Data Primitive	Input	Peer B (input)	The restriction is set to Data Messages only.
Msg Indication B	Msg Datalink Primitive	Output	Peer B (output)	

Table 1-3.1. Datalink Layer: BONeS Ports

3.3.1.1.3. Parameters

The design required parameters for *Address, Bandwidth, Propagation Delay* and *State.*

The *Address*, *Bandwidth* and *Propagation Delay* are BONeS Parameters as they are not expected to be changed during the execution of a simulation. *State* is BONeS Memory, because it is expected to change. An additional *Management Portal* parameter (BONeS Memory) is added to support Management communication. The parameters are listed in Table 1-3.2.

Name	*DS Type*	*Type*	*Design*	*Notes*
Management Portal	Msg Management Set Indication	Memory	Management Message	
Address	INTEGER	Parameter	Address	
Bandwidth	REAL	Parameter	Bandwidth	
Propagation Delay	REAL	Parameter	Propagation Delay	
State	Boolean	Local Memory	State	Default Value is True

Table 1-3.2. Datalink Layer: BONeS Parameters

3.3.1.1.4. Data Structures

The design required a number of Data Structures, implementation detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.3.

Name
IE Datalink Primitive
IE Datalink State
IE Datalink Flow Control
Msg Datalink Primitive
Msg Datalink Connect Primitive
Msg Datalink Disconnect Primitive
Msg Datalink Status Primitive
Msg Datalink Data Primitive
Msg Datalink Connect Indication
Msg Datalink Disconnect Indication
Msg Datalink Data Request
Msg Datalink Data Indication
Msg Datalink Status Indication

Table 1-3.3. Datalink Layer: BONeS Data Structures

3.3.1.1.5. Modules

The top level Datalink Layer Module is shown in Figure 1-3.6. The Ports and Parameters can be seen, along with the top level Transmission Channels, Management and Initialisation. More levels of implementation that are provided in Appendix 2, including the Data Accessors constructed to support the manipulation of Datalink Layer Data Structures.

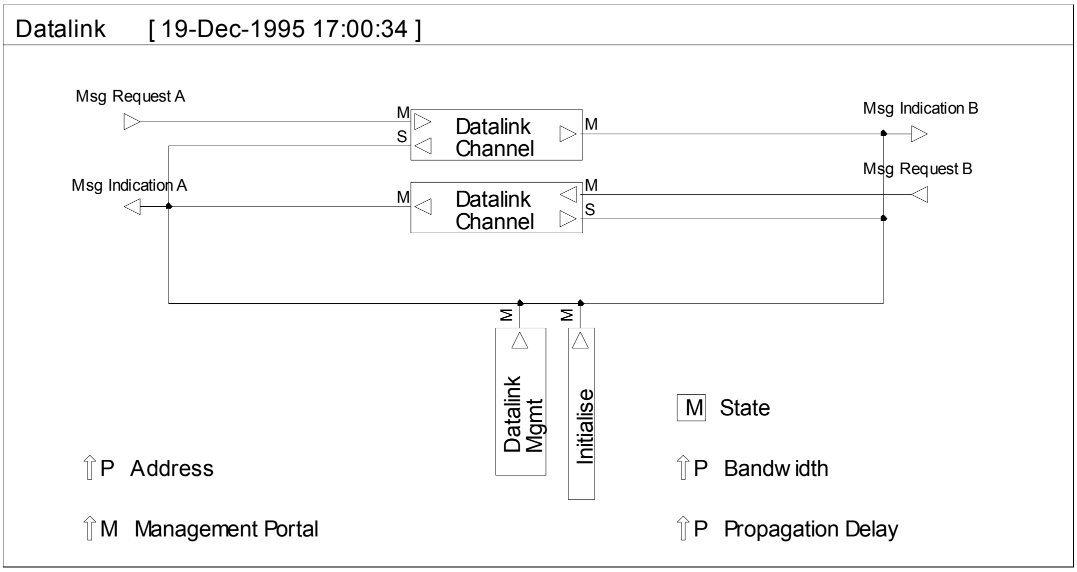


Figure 1-3.6. Datalink Layer Module

3.3.1.2. Network Layer

3.3.1.2.1. Overview

The Network Layer implementation was one of the more complex. The BONeS Modules were constructed as mapped from the design with minor name and tactical alterations. The only architectural alteration of note is that the processing of Datalink Messages was brought up one level. This was done to reduce what seemed to be an unnecessary encapsulation; especially due to the fact that each type of message generated an outwards control signal. This BONeS implementation was straightforward.

The complexity was involved in the construction of the Queue ADT in the 'C' language and its interfaces to BONeS. This involved construction of a low level set of Queue primitives, over which higher-level abstractions were built. Testing was performed on the 'C' language modules independently of BONeS. The ADT was constructed in 'C' for speed efficiency, and to reduce the level of risk in the project.

A first pass construction did implement the Queue in BONeS; but this was revised once it was realised that considerable overhead would result.

3.3.1.2.2. Ports

The design required interfaces for *Upper Layer* and *Datalink Layer*.

Datalink Layer is implemented as a set of input and output ports corresponding to the Datalink Layer's output and input ports respectively. *Upper Layer* is implemented as a set of input and output ports. The ports are listed in Table 1-3.4.

Name	*DS Type*	*Direction*	*Design*	*Notes*
Upper Layer Input	Msg Network Data Primitive	Input	Upper Layer (input)	
Upper Layer Output	Msg Network Primitive	Output	Upper Layer (output)	
Lower Layer Input	Msg Datalink Primitive	Input	Datalink Layer (input)	
Lower Layer Output	Msg Datalink Data Primitive	Output	Datalink Layer (output)	

Table 1-3.4. Network Layer: BONeS Ports

3.3.1.2.3. Parameters

The design required parameters for *Address, End System, Queue Policy* and *Queue Length.*

All of these are parameters are implemented as BONeS Parameters as they are not expected to be changed during the execution of a simulation. The parameters are listed in Table 1-3.5.

Name	*DS Type*	*Type*	*Design*	*Notes*
Address	INTEGER	Parameter	Address	
Queue Discipline	String	Parameter	Queue Policy	Format is specified in the usage notes. Default value is empty.
Queue Length	INTEGER	Parameter	Queue Length	Default value is 50.
End System	Boolean	Parameter	End System	Default value is True.

Table 1-3.5. Network Layer: BONeS Parameters

3.3.1.2.4. Data Structures

The design required a number of Data Structures, implementation detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.6.

Name
IE Network Primitive
IE Network Load Factor
Msg Network Primitive
Msg Network Connect Primitive
Msg Network Disconnect Primitive
Msg Network Status Primitive
Msg Network Data Primitive
Msg Network Connect Indication
Msg Network Disconnect Indication
Msg Network Data Request
Msg Network Data Indication
Msg Network Status Indication

Table 1-3.6. Network Layer: BONeS Data Structures

3.3.1.2.5. Modules

The top level Network Layer Module is shown in Figure 1-3.7. The Ports and Parameters can be seen, along with the top level processing of Datalink Messages and received Network Data Messages. More levels of implementation are provided in Appendix 2, including the Data Accessors constructed to support the manipulation of Network Layer Data Structures.

The top-level implementation corresponds to the design with only one major difference, the expansion of "DFD 1: Process Datalink Message"; this was done as it did not seem beneficial to retain the encapsulation.

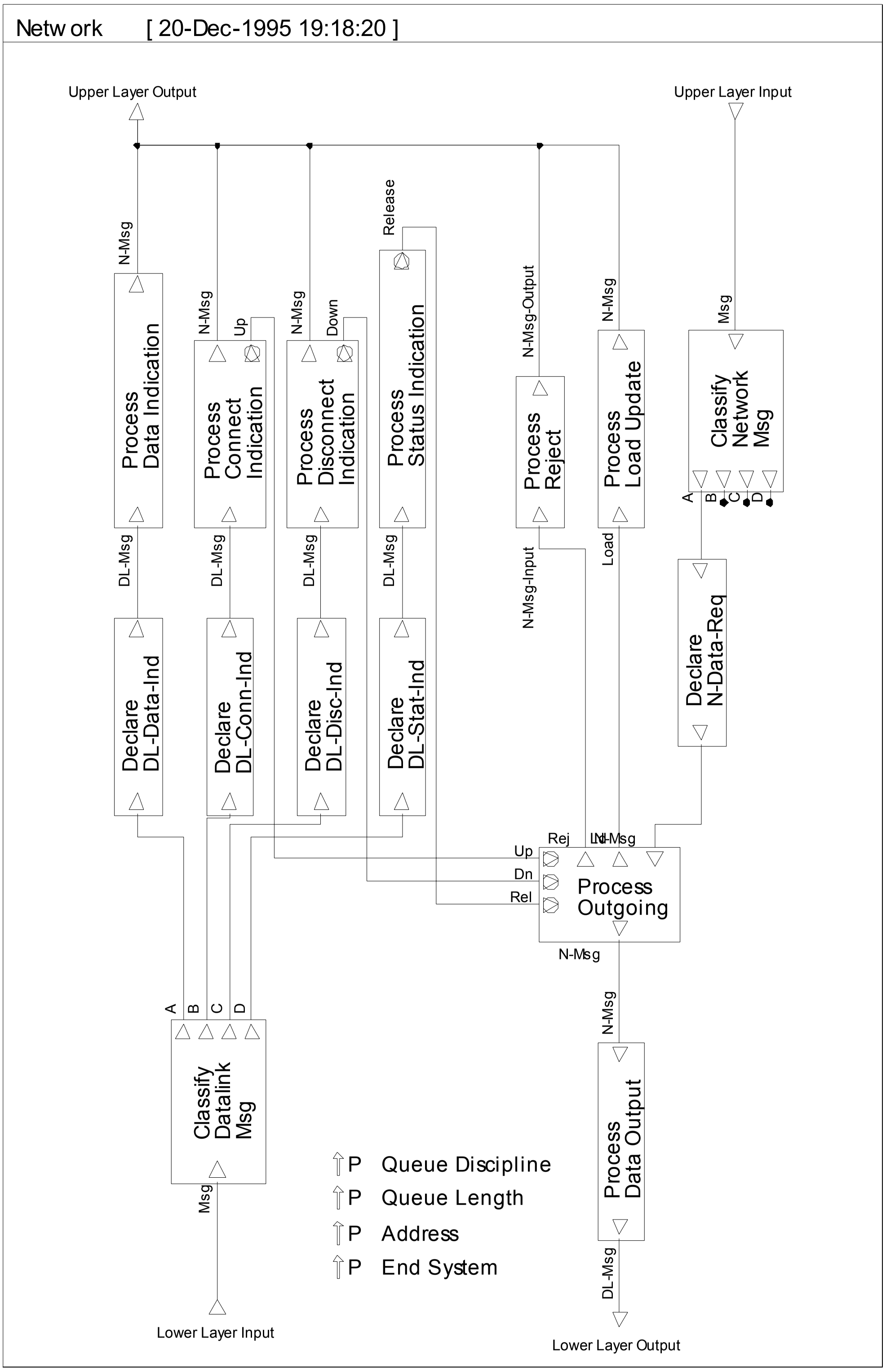


Figure 1-3.7. Network Layer: Module

3.3.1.3. Transport Layer

3.3.1.3.1. Overview

The Transport Layer implementation was the most complex. The BONeS Modules were constructed as mapped from the design with minor name and tactical alterations. This BONeS implementation was straightforward.

The complexity was involved in the construction of the TCP ADT in the 'C' language and its interfaces to BONeS. Testing was performed on the 'C' language modules independently of BONeS. The ADT was constructed in 'C' for speed efficiency, and to reduce the level of risk in the project.

A first pass construction did start to implement TCP in BONeS; but complexity ensued, and the processing of fragments indicated that a lower level language would be more practical. Time performance was also considered an issue.

3.3.1.3.2. Ports

The design required interfaces for *Upper Layer, Network Layer* and *Management.*

Management is implemented using the "portal" mechanism described as part of the Management implementation. *Network Layer* is implemented as a set of input and output ports corresponding to the Network Layer's output and input ports respectively. *Upper Layer* is implemented as a set of input and output ports. The ports are listed in Table 1-3.7.

Name	*DS Type*	*Direction*	*Design*	*Notes*
Upper Layer Input	Msg Transport Primitive	Input	Upper Layer (input)	
Upper Layer Output	Msg Transport Primitive	Output	Upper Layer (output)	
Lower Layer Input	Msg Network Primitive	Input	Network Layer (input)	
Lower Layer Output	Msg Network Data Primitive	Output	Network Layer (output)	

Table 1-3.7. Transport Layer: BONeS Ports

3.3.1.3.3. Parameters

The design required parameters for *Address, Initial Sequence Number* and *Destination Address.*

The *Address* is a BONeS Parameter as it is not expected to be changed during the execution of a simulation. *Initial Sequence Number* and *Destination Address are* BONeS Memory, because they are expected to change. *State* is an internal parameter (BONeS Memory), used to retain knowledge about the current connection state. An additional *Management Portal* parameter (BONeS Memory) is added to support Management communication. The parameters are listed in Table 1-3.8.

Name	*DS Type*	*Type*	*Design*	*Notes*
Management Portal	Msg Management Set Indication	Memory	Management Message	
Address	INTEGER	Parameter	Address	
Initial Sequence Number	INTEGER	Memory	Initial Sequence Number	
Destination Address	INTEGER	Memory	Destination Address	
State	Boolean	Local Memory		Default value is False. Indicates whether TCP is connected.

Table 1-3.8. Transport Layer: BONeS Parameters

3.3.1.3.4. Data Structures

The design required a number of Data Structures, implementation detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.9.

Name
IE Transport Primitive
IE Transport Parameters
Msg Transport Primitive
Msg Transport Connect Primitive
Msg Transport Disconnect Primitive
Msg Transport Data Primitive
Msg Transport Connect Request
Msg Transport Disconnect Request
Msg Transport Data Request
Msg Transport Data Indication
Msg Transport TCP

Table 1-3.9. Transport Layer: BONeS Data Structures

3.3.1.3.5. Modules

The top level Transport Layer Module is shown in Figure 1-3.8. The Ports and Parameters can be seen, along with the top-level Connection Manager and TCP Established Processing, which has wrapper Transport Interface and Network Interface modules. More levels of implementation are provided in Appendix 2, including the Data Accessors constructed to support the manipulation of Transport Layer Data Structures.

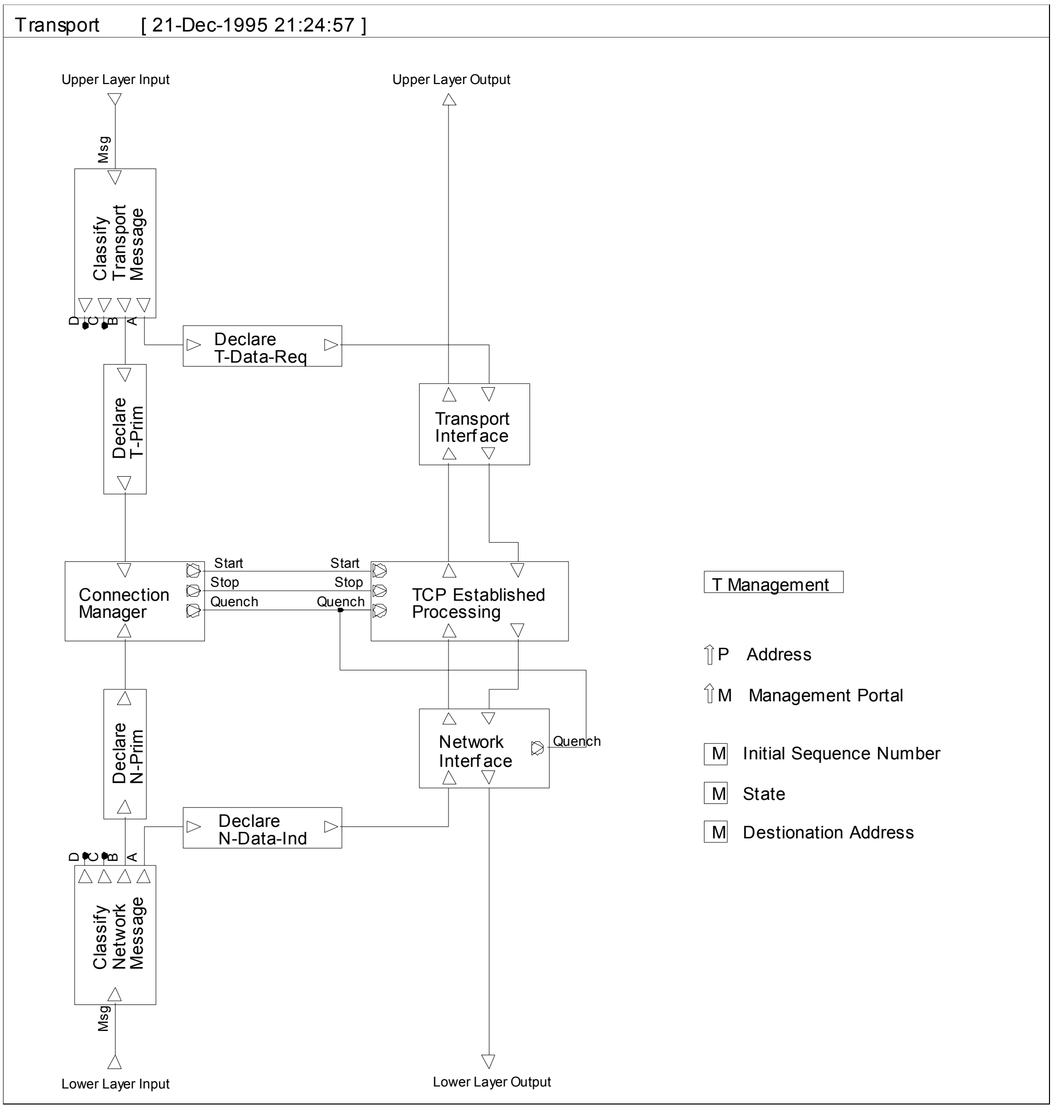


Figure 1-3.8. Transport Layer: Module

3.3.1.4. Network-Adaption Layer

3.3.1.4.1. Overview

The implementation mapping from the design was straightforward, and apart from minor name and tactical alterations, there are no significant changes.

3.3.1.4.2. Ports

The design required interfaces for *Upper Layer, Network Layer* and *Management.*

Management is implemented using the "portal" mechanism described as part of the Management implementation. *Network Layer* is implemented as a set of input and output ports corresponding to the Network Layer's output and input ports respectively. *Upper Layer* is implemented as a set of input and output ports. The ports are listed in Table 1-3.10.

Name	*DS Type*	*Direction*	*Design*	*Notes*
Data-Length	INTEGER	Input	Upper Layer (input)	
N-Msg Input	Msg Network Primitive	Input	Network Layer (input)	
N-Msg Output	Msg Network Data Primitive	Output	Network Layer (output)	

Table 1-3.10. Network-Adaption Layer: BONeS Ports

3.3.1.4.3. Parameters

The design required parameters for *Address* and *Address List.*

The *Address* is a BONeS Parameter as it is not expected to be changed during the execution of a simulation. *Address List* is BONeS Memory, because it is expected to change. *Network State* is an internal parameter (BONeS Memory), used to retain knowledge about the current state of the Network Layer. An additional *Management Portal* parameter (BONeS Memory) is added to support Management communication. The parameters are listed in Table 1-3.11.

Name	*DS Type*	*Type*	*Design*	*Notes*
Management Portal	Msg Management Set Indication	Memory	Management Message	
Address	INTEGER	Parameter	Address	
Address List	INT-VECTOR	Memory	Address List	
Network State	Boolean	Local Memory		Default value is False. Indicates whether Network Layer is connected.

Table 1-3.11. Network-Adaption Layer: BONeS Parameters

3.3.1.4.4. Data Structures

The design required a number of Data Structures, detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.12.

Name
IE Network-Adaption Primitive
IE Network-Adaption Address List

Table 1-3.12. Network-Adaption Layer: BONeS Data Structures

3.3.1.4.5. Modules

The top level Network-Adaption Layer Module is shown in Figure 1-3.9. The Ports and Parameters can be seen, along with the top level processing of Network Layer input and output. More levels of implementation are provided in Appendix 2, including the Data Accessors constructed to support the manipulation of Network-Adaption Layer Data Structures.

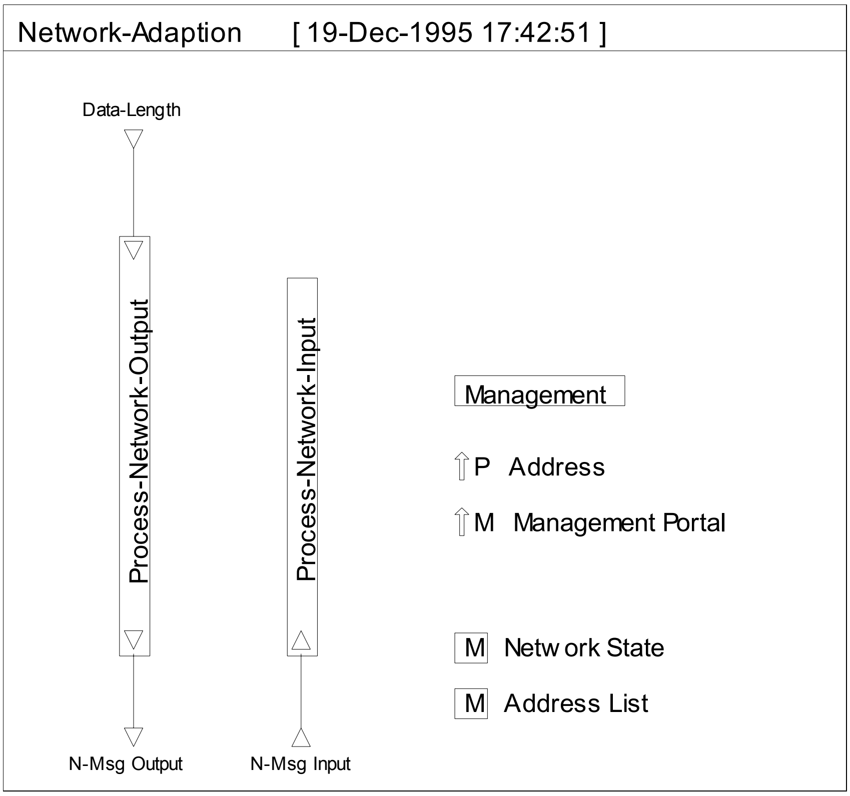


Figure 1-3.9. Network-Adaption Layer Module

3.3.1.5. Transport-Adaption Layer

3.3.1.5.1. Overview

The implementation mapping from the design was straightforward, and apart from minor name and tactical alterations, there are no significant changes.

3.3.1.5.2. Ports

The design required interfaces for *Upper Layer, Transport Layer* and *Management.*

Management is implemented using the "portal" mechanism described as part of the Management implementation. *Transport Layer* is implemented as a set of input and output ports corresponding to the Transport Layer's output and input ports respectively. *Upper Layer* is implemented as a set of input and output ports. The ports are listed in Table 1-3.13.

Name	*DS Type*	*Direction*	*Design*	*Notes*
Data-Length	INTEGER	Input	Upper Layer (input)	
Msg-In	Msg Transport Primitive	Input	Transport Layer (input)	
Msg-Out	Msg Transport Primitive	Output	Transport Layer (output)	

Table 1-3.13. Transport-Adaption Layer: BONeS Ports

3.3.1.5.3. Parameters

The design required parameters for an *Address.*

The *Address* is a BONeS Parameter as it is not expected to be changed during the execution of a simulation. An additional *Management Portal* parameter (BONeS Memory) is added to support Management communication. The parameters are listed in Table 1-3.14.

Name	*DS Type*	*Type*	*Design*	*Notes*
Management Portal	Msg Management Set Indication	Memory	Management Message	
Address	INTEGER	Parameter	Address	

Table 1-3.14. Transport-Adaption Layer: BONeS Parameters

3.3.1.5.4. Data Structures

The design required a number of Data Structures, detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.15.

Name
IE Transport-Adaption Primitive
IE Transport-Adaption Connect
IE Transport-Adaption Disconnect

Table 1-3.15. Transport-Adaption Layer: BONeS Data Structures

3.3.1.5.5. Modules

The top level Transport-Adaption Layer Module is shown in Figure 1-3.10. The Ports and Parameters can be seen, along with the top level processing of Transport Layer input and output. More levels of implementation are provided in Appendix 2, including the Data Accessors constructed to support the manipulation of Transport-Adaption Layer Data Structures.

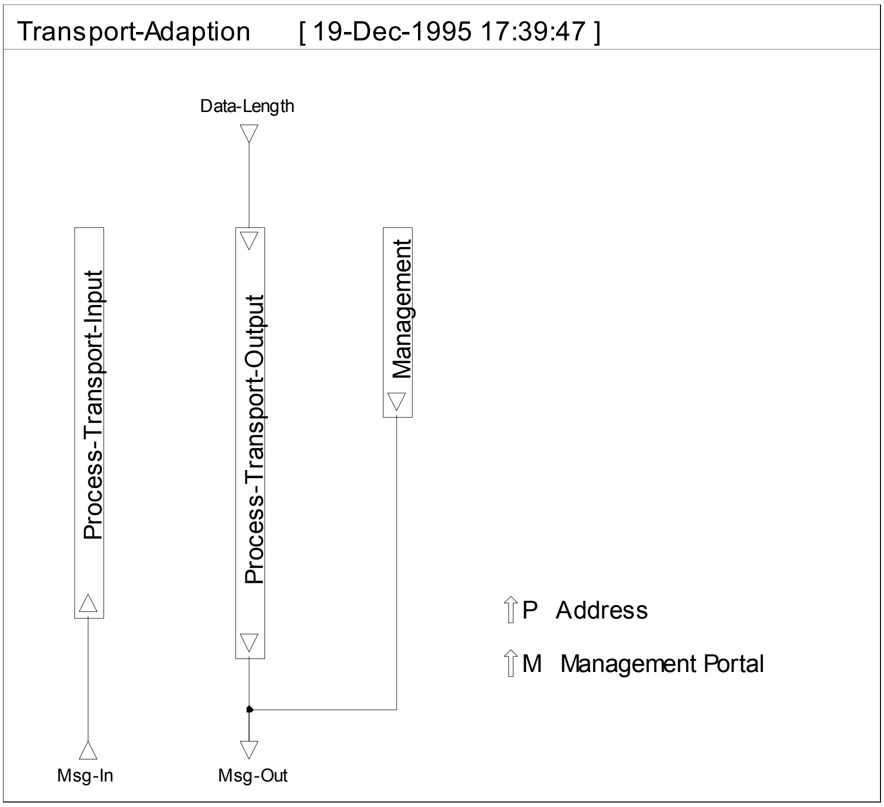


Figure 1-3.10. Transport-Adaption Layer Module

3.3.1.6. Routing-Module

3.3.1.6.1. Overview

The implementation mapping from the design was straightforward, and apart from minor name and tactical alterations, there are no significant changes of interest.

There were two significant points in the implementation that did require more detailed consideration. The first was the construction of the Next Hop computation involving an iterative technique. This perhaps should have been constructed in a lower level language, as it would tend to be a bottleneck procedure in the system.

The second issue is the relationship between the Routing-Switch and the "physical" switch leading to each individual Network Interface. The two were decoupled so that additional Network Interfaces could be supported without alteration of the Routing-Switch. The issue is that it must be ensured that the destination output interface reaches the "physical" switch between the output Message. This is achieved by using a shared Memory variable.

3.3.1.6.2. Ports

The design required interfaces for *Network Layers* and *Management.*

Management is implemented using the "portal" mechanism described as part of the Management implementation. *Network Layer* is implemented as a set of input and output ports corresponding to the Network Layer's output and input ports respectively. There are five of these available, and construction is such that addition of more interfaces is trivial. The ports are listed in Table 1-3.16.

Name	*DS Type*	*Direction*	*Design*	*Notes*
In-A	Msg Network Primitive	Input	Network Layer A (input)	
Out-A	Msg Network Data Primitive	Output	Network Layer A (output)	
...	...	...	...	
In-E	Msg Network Primitive	Input	Network Layer E (input)	
Out-E	Msg Network Data Primitive	Output	Network Layer E (output)	

Table 1-3.16. Routing-Module: BONeS Ports

3.3.1.6.3. Parameters

The design required parameters for *Address* and *Routing-Table Entries.*

The *Address* is a BONeS Parameter as it is not expected to be changed during the execution of a simulation. *Routing Table* is BONeS Memory, because it is expected to change. *Maximum Interfaces* is an internal parameter (BONeS Parameter), used to retain knowledge about the number of interfaces. *Interface Load Status, Interface Availability Status* and *Output Interface* are internal parameters (BONeS Memory), used to retain knowledge about associated Network Layer status. An additional *Management Portal* parameter (BONeS Memory) is added to support Management communication. The parameters are listed in Table 1-3.17.

Name	*DS Type*	*Type*	*Design*	*Notes*
Management Portal	Msg Management Set Indication	Memory	Management Message	
Address	INTEGER	Parameter	Address	
Routing Table	REAL-MATRIX	Memory	Routing-Table Entries	
Maximum Interfaces	INTEGER	Parameter		Default value is 5.
Interface Load Status	REAL-VECTOR	Memory		Default values are 0.0.
Interface Availability Status	INT-VECTOR	Memory		Default values are False.

Table 1-3.17. Routing-Module: BONeS Parameters

3.3.1.6.4. Data Structures

The design required a number of Data Structures, detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.18.

Name
IE Routing-Module Primitive
IE Routing-Module Route-Entry

Table 1-3.18. Routing-Module: Data Structures

3.3.1.6.5. Modules

The top level Routing-Module Module is shown in Figure 1-3.11. The Ports and Parameters can be seen, along with the top level Routing-Switch and Network Interface modules. The Routing-Switch utilises the 8-Way Switch to guide an output data structure along the fabric to the correct Network Layer. More levels of implementation are provided in Appendix 2, including the Data Accessors constructed to support the manipulation of Routing-Module Data Structures.

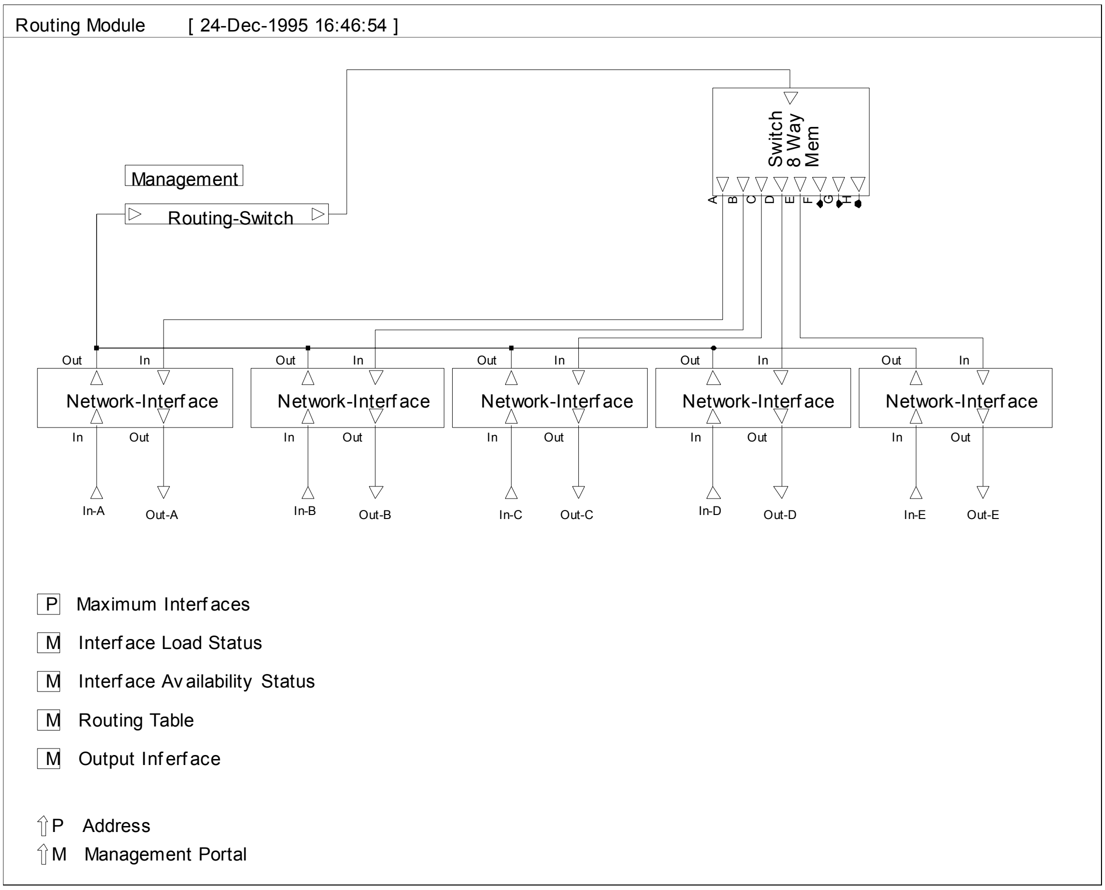


Figure 1-3.11. Routing-Module Module

3.3.1.7. Generator

3.3.1.7.1. Overview

The implementation mapping from the design was straightforward, and apart from minor name and tactical alterations, there are no significant changes of interest.

This Module does use a ‘C’ implemented library for the creation of data values, but the interfacing for this was trivial.

3.3.1.7.2. Ports

The design required interfaces for *Lower Layer* and *Management.*

Management is implemented using the “portal” mechanism described as part of the Management implementation. *Lower Layer* is an output only port. The ports are listed in Table 1-3.19.

Name	*DS Type*	*Direction*	*Design*	*Notes*
Data-Length	INTEGER	Output	Lower Layer (output)	

Table 1-3.19. Generator: BONeS Ports

3.3.1.7.3. Parameters

The design required parameters for an *Address.*

The *Address* is a BONeS Parameter as it is not expected to be changed during the execution of a simulation. *Generator Timer* is an internal parameter (BONeS Event), used to retain knowledge about the timers set up by the Generator. An additional *Management Portal* parameter (BONeS Memory) is added to support Management communication. The parameters are listed in Table 1-3.20.

Name	*DS Type*	*Type*	*Design*	*Notes*
Management Portal	Msg Management Set Indication	Memory	Management Message	
Address	INTEGER	Parameter	Address	

Table 1-3.20. Generator: BONeS Parameters

3.3.1.7.4. Data Structures

The design required a number of Data Structures, detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.21.

Name
IE Generator Primitive
IE Generator Setup Primitive
IE Generator Stop
IE Generator Setup TCP
IE Generator Setup Telnet
IE Generator Setup Statistical

Table 1-3.21. Generator: BONeS Data Structures

3.3.1.7.5. Modules

The top level Generator Module is shown in Figure 1-3.12. The Ports and Parameters can be seen, along with the top level processing of Setup and Stop actions. More levels of implementation are provided in Appendix 2, including the Data Accessors constructed to support the manipulation of Generator Data Structures.

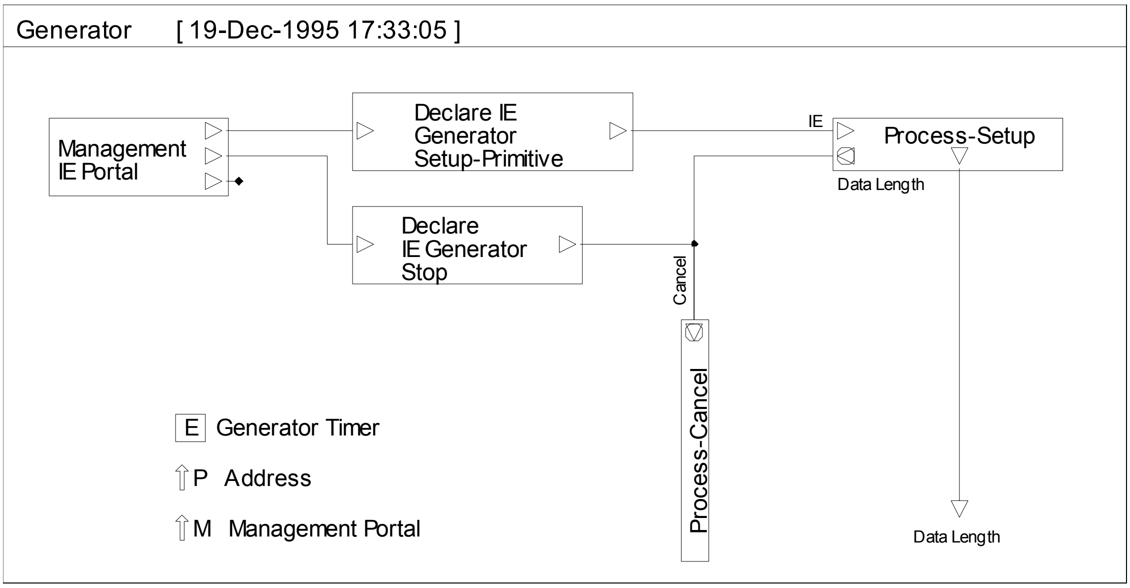


Figure 1-3.12. Generator Module

3.3.1.8. Management

3.3.1.8.1. Overview

The implementation mapping from the design was straightforward, and apart from minor name and tactical alterations, there are no significant changes of interest. This Module is the heart of the simulation, in that it interprets commands placed into a management file and converts these into Information Elements that are sent to specifically addressed Modules.

3.3.1.8.2. Ports

The design required interfaces for *Modules* and *Initialisation.*

Modules is implemented using a "portal" mechanism which involves a common shared memory location used by all Modules requiring Management control. A Message is placed into the shared memory and read by the Module with an *Address* equivalent to the destination address in the Message. *Initialisation* is a trigger mechanism used to indicate that the simulation has started.

3.3.1.8.3. Parameters

The design required parameters for a *Filename.*

The *Filename* is a BONeS Parameter as it is not expected to be changed during the execution of a simulation. *Address* and *File* are internal parameters (BONeS Memory), used to retain knowledge about the destination address for a Management Message and the active file handle from which commands are being read. An additional *Management Portal* parameter (BONeS Memory) is added to support Management communication (in this case, it is distinctly write). The parameters are listed in Table 1-3.22.

Name	*DS Type*	*Type*	*Design*	*Notes*
Management Portal	Msg Management Set Indication	Memory	Management Message	
Filename	String	Parameter	Filename	
File	FILE	Memory		
Address	INTEGER	Memory		

Table 1-3.22. Management: BONeS Parameters

3.3.1.8.4. Data Structures

The design required a number of Data Structures, detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.23.

Name
Msg Management Primtive
Msg Management Set Primitive
Msg Management Set Indication

Table 1-3.23. Management: BONeS Data Structures

3.3.1.8.5. Modules

The top level Management Module is shown in Figure 1-3.13. The Ports and Parameters can be seen, along with the top level processing. This top level processing involves an event loop that reads a command, waits and then executes it. More levels of implementation are provided in Appendix 2, including the Data Accessors constructed to support the manipulation of Management Data Structures.

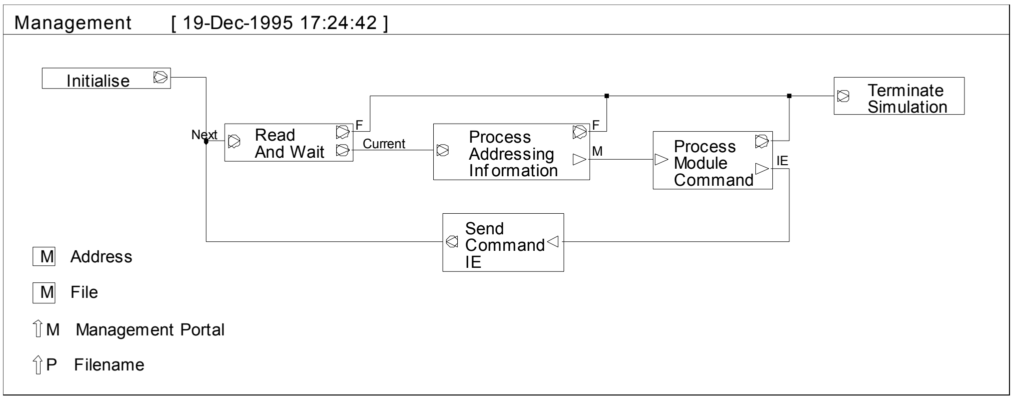


Figure 1-3.13. Management Module

3.3.2. Miscellaneous Modules

3.3.2.1. Statistical Parameter

3.3.2.1.1. Overview

The Statistical Parameter was implemented by translating each function into a Module. For the purposes of creating Statistical Parameters, the defined Data Structures can be created individually. For computing a value according to a Statistical Parameter, there is a single module that hides the implementation.

3.3.2.1.2. Data Structures

There are Data Structures defined to encapsulate each type of Statistical Parameter, derived from a base Statistical Parameter. Detail for these is provided in Appendix 2. The Data Structures are listed in Table 1-3.24.

Name
Statistical Parameter
Statistical Parameter Constant
Statistical Parameter Uniform
Statistical Parameter Normal
Statistical Parameter Poisson
Statistical Parameter Exponential

Table 1-3.24. Statistical Parameter: BONeS Data Structures

3.3.2.1.3. Modules

To create particular Statistical Parameters, as required in the design, BONeS Data Structure creation block are used with the Statistical Parameter of interest. There are no explicit Modules constructed for this functionality.

To compute a value for a particular Statistical Parameter, a single Module is defined. The top level Module is shown in FIGURE. Internally, individual Modules execute functionality related to particular Statistical Parameters; this level of detail is provided in Appendix 2.

3.3.2.2. Transport Layer TCP Probe

3.3.2.2.1. Overview

The Probe was implemented through a top level Module and a 'C' implementation, the specific details of which are provided in Appendix 2.

3.3.2.2.2. Ports

The design required interfaces for the TCB Number and the output variable. The ports are listed in Table 1-3.25.

Name	*DS Type*	*Direction*	*Notes*
TCB Number	INTEGER	Input	Provides the TCB Number for locating the datum.
Value	REAL	Output	Provides the actual datum collected.

Table 1-3.25. Transport Layer TCP Probe: Ports

3.3.2.2.3. Parameters

The design required two parameters, for *Changed Only* and *Data Type,* both of which are implemented as BONeS Parameters. The parameters are listed in Table 1-3.26, and Table 1-3.27.

Name	*DS Type*	*Type*	*Notes*
Changed Only	Boolean	Parameter	Indicates whether only changed values should be output
Type	STRING	Parameter	Indicates the particular datum that is to be collected.

Table 1-3.26. Transport Layer TCP Probe: BONeS Parameters

Name	*Notes*
Congestion Window	
Slow Start Threshold	
Retransmission Events	
Round Trip Time Average	The smoothed RTT computed by TCP.
Round Trip Time Variance	As computed by TCP.
Send Window	
Unacknowledged Data	Computed from window state values.
Timer Expiries	
Acknowledgements Received	The acknowledgement value.
KB Retransmitted	
KB Transmitted	
Reassembly Queue Size	Number of packets, not byte count.

Table 1-3.27. Transport Layer TCP Probe: Types

3.3.2.2.4. Modules

There is a single Module, which is displayed (along with corresponding 'C' source) in Appendix 2.

3.3.2.3. Network Layer Queue Probe

3.3.2.3.1. Overview

The Probe was implemented through a top level Module and a ‘C’ implementation, the specific details of which are provided in Appendix 2.

3.3.2.3.2. Ports

The design required interfaces for the Queue Number and the output variable. The ports are listed in Table 1-3.28.

Name	*DS Type*	*Direction*	*Notes*
Queue Number	INTEGER	Input	The Queue to be looked at.
Value	REAL	Output	The data variable computed.

Table 1-3.28. Network Layer Queue Probe: Ports

3.3.2.3.3. Parameters

The design required two parameters, for *Changed Only* and *Data Type,* both of which are implemented as BONeS Parameters. The parameters are listed in Table 1-3.29 and Table 1-3.30.

Name	*DS Type*	*Type*	*Notes*
Changed Only	Boolean	Parameter	Whether or not to provide changed values only.
Type	STRING	Parameter	The type of datum to be computed.
Address	INTEGER	Parameter	Used to indicate addresses in the Queue to be looked at.

Table 1-3.29. Network Layer Queue Probe: BONeS Parameters

Name	*Notes*
Size Src Address Count Dst Address Count	

Table 1-3.30. Network Layer Queue Probe: Types

3.3.2.3.4. Modules

There is a single Module, which is displayed (along with corresponding ‘C’ source) in Appendix 2.

3.3.2.4. Common

3.3.2.4.1. Overview

There are a number of common modules that have been implemented, the detail for which is given in Appendix 2. A summary is provided in Table 1-3.31.

Module	*Description*
Boolean ==	Compare Boolean Data Structures.
Create Msg Application Data	Create a Msg Application Data of given Length.
Extract Msg Application Data	Extract the Length and Creation Time of a Msg Application Data.
IE Switch	Switch an IE onto multiple output ports depending upon the type of IE.
Msg Switch	Switch a Msg onto multiple output ports depending upon the type of Msg, with 4 output ports.
Switch 8-Way Mem	Same as Msg Switch, but this time has 8 output ports.
Type == Switch	Compare a Data Structure's type to another type.

Table 1-3.31. Common Modules

The Data Structures are listed in Table 1-3.32, more detail is given in Appendix 2.

Name
Msg Primitive
Msg Application Primitive
Msg Application Data
Boolean

Table 1-3.32. Common Data Structures

3.4. Components

As designed, the Components are constructed by aggregating the Primary Modules. Each Module uses has a number of arguments, these are either set to a constant value, or exported so that they can be set when using the Component. Here, both the construction of the Components and the arguments used for consistent Modules are shown.

3.4.1. Host

The Host aggregates a Generator, Transport-Adaption Layer, Transport Layer and Network Layer to achieve its goal of being a TCP capable Network End System. The construction is shown in Figure 1-3.14. All argument values are shown in Table 1-3.33.

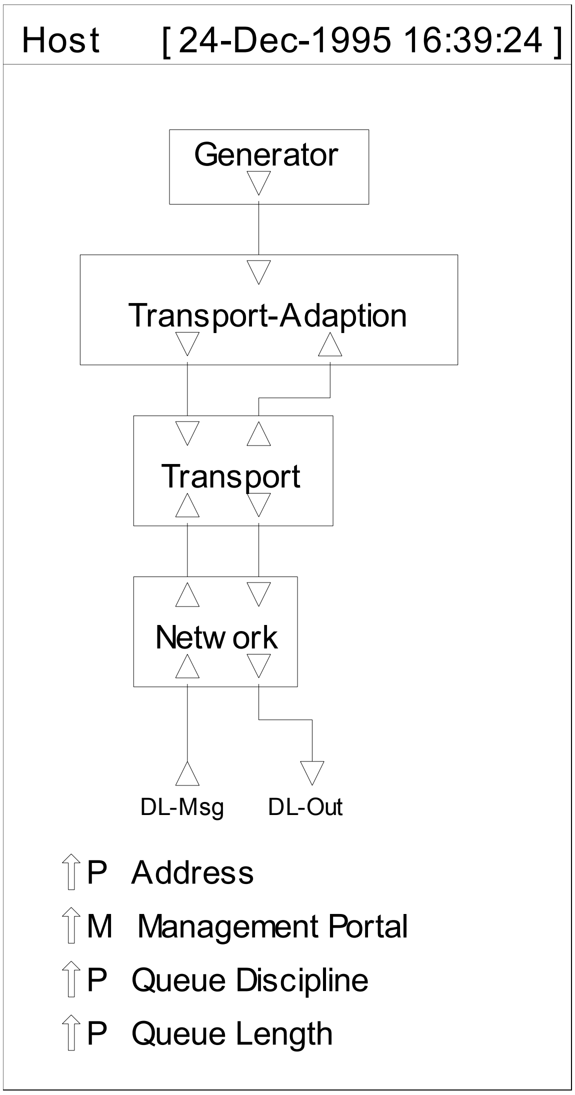


Figure 1-3.14. Host Component

Module	*Name*	*DS Type*	*Value*
Generator, Transport-Adaption Layer, Transport Layer, Network Layer	Management Portal	Msg Management Set Indication	Exported. No default.
Generator, Transport-Adaption Layer, Transport Layer, Network Layer	Address	INTEGER	Exported. No default.
Network Layer	Queue Discipline	String	Exported. Default is empty.
Network Layer	Queue Length	INTEGER	Exported. Default is 20.
Network Layer	End System	Boolean	True

Table 1-3.33. Host: Parameters

3.4.2. Traffic

The Traffic aggregates a Generator, Network-Adaption Layer and Network Layer to achieve its goal of being a Network End System capable of transmitting and receiving traffic of defined characteristic. The construction is shown in Figure 1-3.15. All argument values are shown in Table 1-3.34.

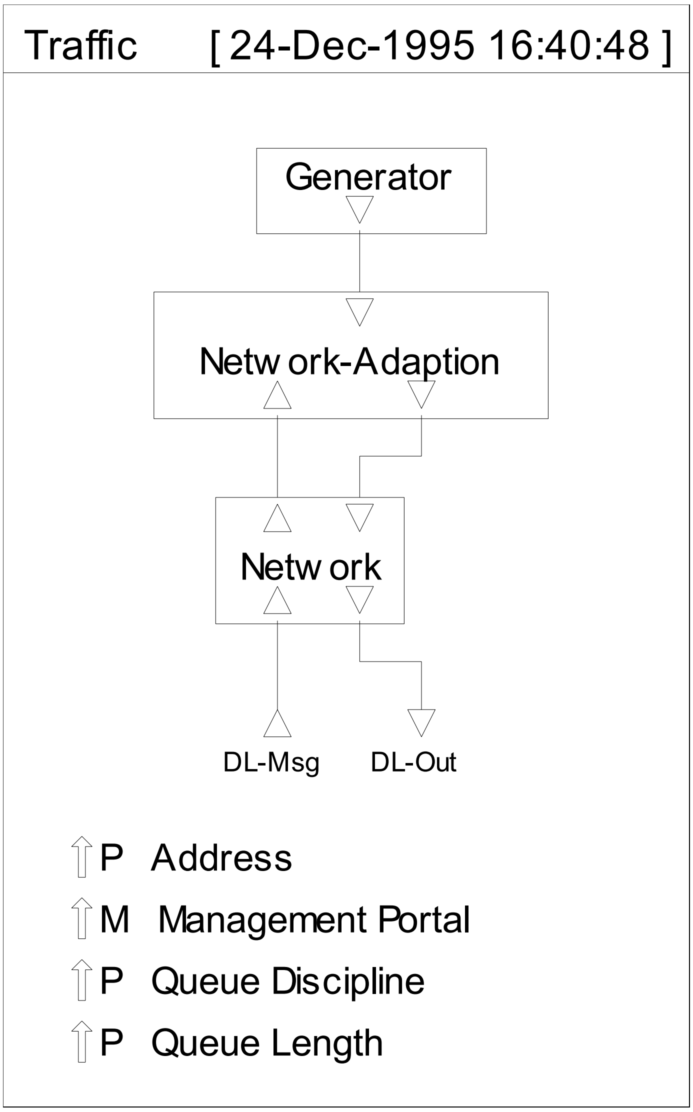


Figure 1-3.15. Traffic Component

Module	*Name*	*DS Type*	*Value*
Generator, Network-Adaption Layer, Network Layer	Management Portal	Msg Management Set Indication	Exported. No default.
Generator, Network-Adaption Layer, Network Layer	Address	INTEGER	Exported. No default.
Network Layer	Queue Discipline	String	Exported. Default is empty.
Network Layer	Queue Length	INTEGER	Exported. Default is 20.
Network Layer	End System	Boolean	True

Table 1-3.34. Traffic: Parameters

3.4.3. Link

The Link consists only of a Datalink Layer. The construction is shown in Figure 1-3.16. All argument values are shown in Table 1-3.35.

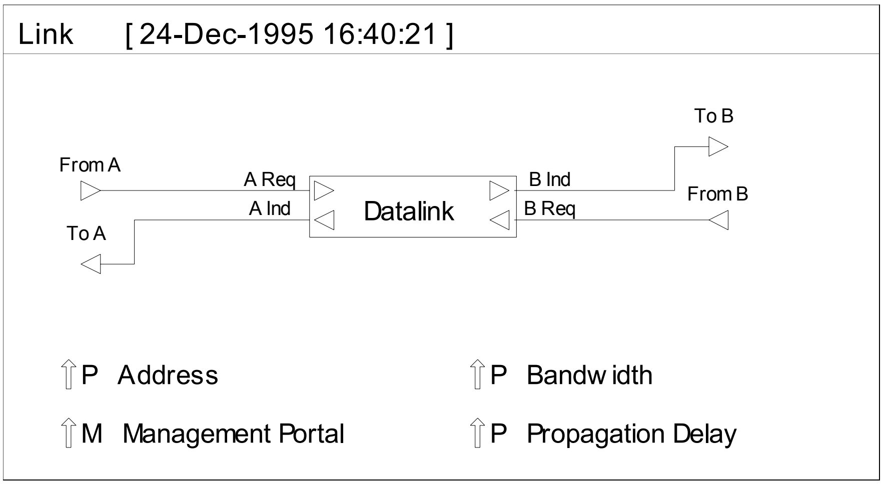


Figure 1-3.16. Link Component

Module	*Name*	*DS Type*	*Value*
Datalink Layer	Management Portal	Msg Management Set Indication	Exported. No default.
Datalink Layer	Address	INTEGER	Exported. No default.
Datalink Layer	Bandwidth	INTEGER	Exported. Default is 64000.
Datalink Layer	Propagation Delay	REAL	Exported. Default is 0.007

Table 1-3.35. Link: Parameters

3.4.4. Router

The Router is constructed by aggregating a single Routing-Module with a number of Network Layers. In this case, note that the Network Layers are all given the same Address. The construction is shown in Figure 1-3.17. All argument values are shown in Table 1-3.36.

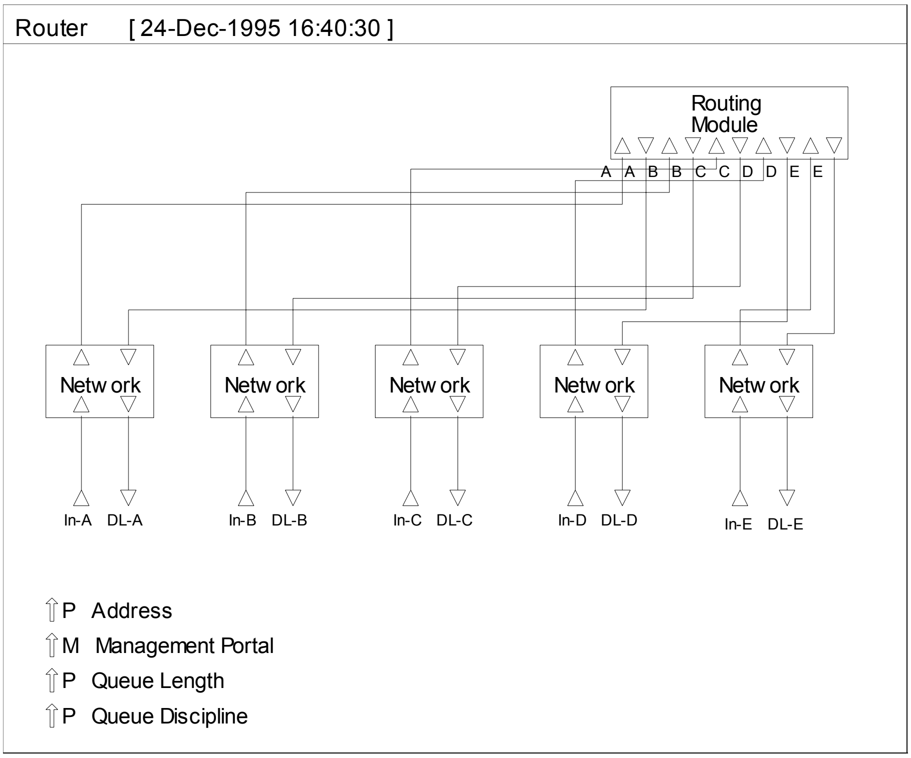


Figure 1-3.17. Router Component

Module	*Name*	*DS Type*	*Value*
Routing-Module, Network Layer(s)	Management Portal	Msg Management Set Indication	Exported. No default.
Routing-Module, Network Layer(s)	Address	INTEGER	Exported. No default.
Network Layer(s)	Queue Discipline	String	Exported. Default is empty.
Network Layer(s)	Queue Length	INTEGER	Exported. Default is 20.
Network Layer(s)	End System	Boolean	False

Table 1-3.36. Router: Parameters

3.4.5. LAN

The LAN builds upon the basic Components to provide a model of a simple Local Area Network. In this model, there are three Hosts, each of which is connected to a Router. The Router is connected to output ports of the LAN. Therefore, the LAN can be connected to another LAN, or Router or whatever, via. a Link. The chosen arguments are representative of a CSMA/CD (Ethernet) LAN.

The construction is shown in Figure 1-3.18. All argument values are shown in Table 1-3.37.

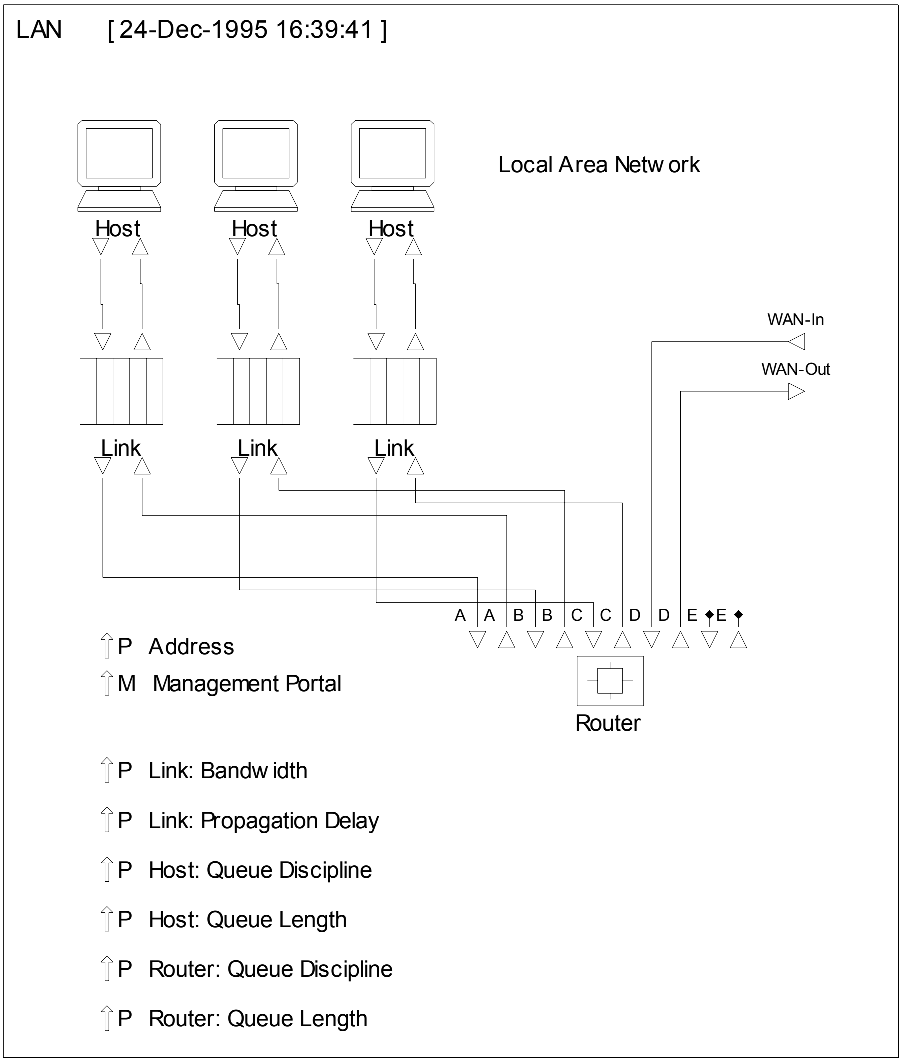


Figure 1-3.18. LAN Component

Module	*Name*	*DS Type*	*Value*
Hosts, Links, Router	Management Portal	Msg Management Set Indication	Exported. No default.
Router	Address	INTEGER	Exported. No default.
Hosts (1 ... 3)	Address	INTEGER	Router Address + (Host Number)
Links (1 ... 3)	Address	INTEGER	Router Address + (Link Number) + 5
Links	Bandwidth	INTEGER	Exported. Default is 1000000.
Links	Propagation Delay	REAL	Exported. Default is 0.001
Hosts	Queue Discipline	String	Exported. Default is empty.
Hosts	Queue Length	INTEGER	Exported. Default is 10.
Router	Queue Discipline	String	Exported. Default is empty.
Router	Queue Length	INTEGER	Exported. Default is 20.

Table 1-3.37. LAN: Parameters

3.4.6. LAN -- Traffic

The LAN -- Traffic is similar to the LAN, however it has a Traffic End System in place of a Host End System.

The construction is shown in Figure 1-3.19. All argument values are shown in Table 1-3.38.

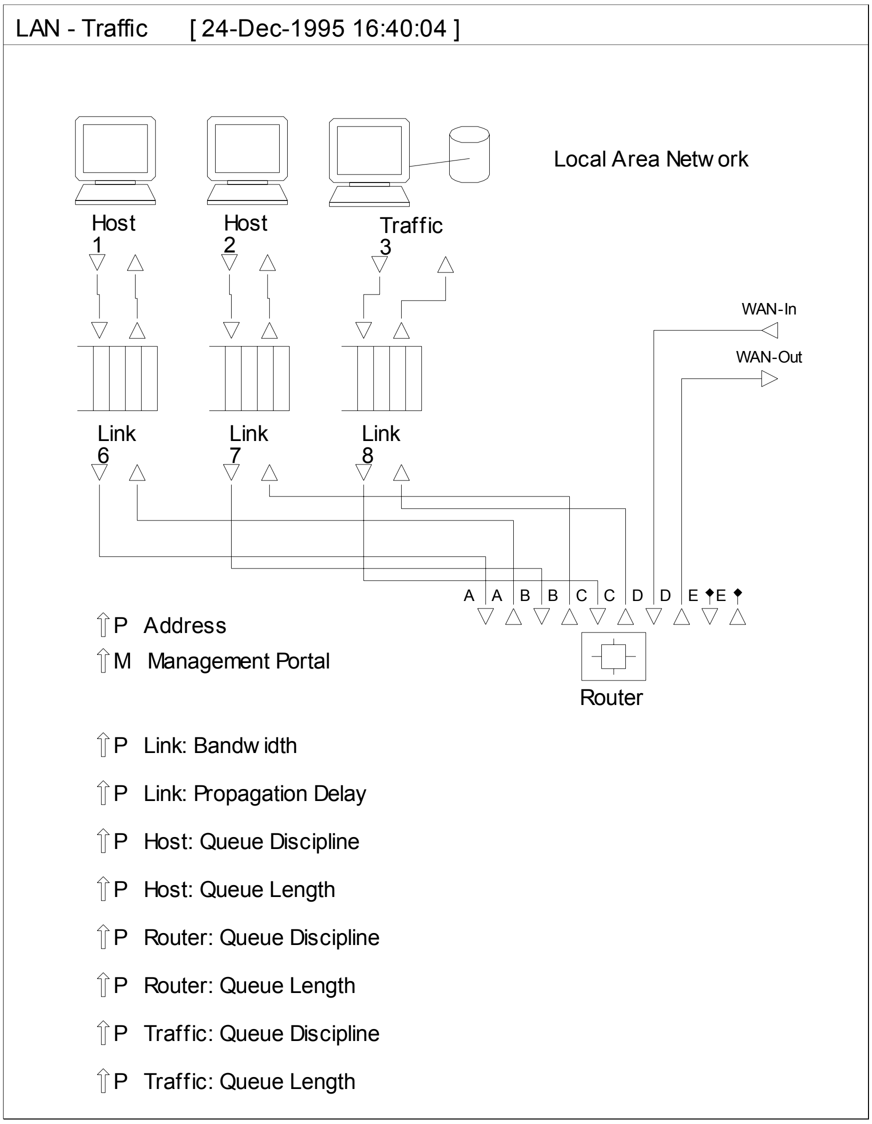


Figure 1-3.19. LAN -- Traffic Component

Module	*Name*	*DS Type*	*Value*
Hosts, Links, Router	Management Portal	Msg Management Set Indication	Exported. No default.
Router	Address	INTEGER	Exported. No default.
Hosts (1 ... 2)	Address	INTEGER	Router Address + (Host Number)
Traffic	Address	INTEGER	Router Address + 3
Links (1 ... 3)	Address	INTEGER	Router Address + (Link Number) + 5
Links	Bandwidth	INTEGER	Exported. Default is 1000000.
Links	Propagation Delay	REAL	Exported. Default is 0.001
Hosts	Queue Discipline	String	Exported. Default is empty.
Hosts	Queue Length	INTEGER	Exported. Default is 10.
Traffic	Queue Discipline	String	Exported. Default is empty.
Traffic	Queue Length	INTEGER	Exported. Default is 10.
Router	Queue Discipline	String	Exported. Default is empty.
Router	Queue Length	INTEGER	Exported. Default is 20.

Table 1-3.38. LAN -- Traffic: Parameters

3.4.7. Simulation Management

The Simulation Management consists only of a Management Module. The construction is shown in Figure 1-3.20. All argument values are shown in Table 1-3.39.

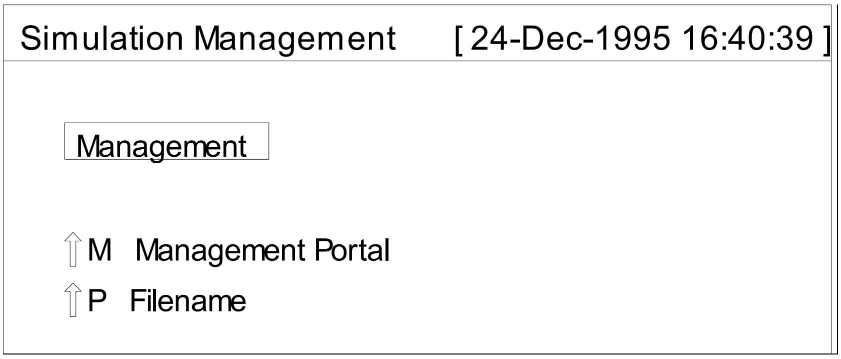


Figure 1-3.20. Simulation Management Component

Module	*Name*	*DS Type*	*Value*
Management	Management Portal	Msg Management Set Indication	Exported. No default.
Management	Filename	String	Exported. No default.

Table 1-3.39. Simulation -- Management: Parameters

3.5. Usage Notes

When constructing a Simulation using the Components given here, there are a number of issues that must be addressed. All Components can be linked with one another, provided that corresponding ports match. The Data Structure limitations on Component ports provide a pre-condition mechanism for assuring that Components are correctly connected. For example, it is not possible to connect the Datalink Layer to the Transport Layer. This clearly justifies are design decisions in relation to the use of such primitives.

Components have arguments that must be specified, these arguments are generally related to the particular Component itself, but there are a few arguments that are present on all Components. The first of these is an “Address”. The Address uniquely identifies a participating Component in the Simulation, and is used for addressing Messages and for addressing Management. A unique Address must be supplied for each Component.

Rather than complicate the issue and require connections between Management and every Component, a portal mechanism is used; this involves Management Messages being transferred from the Simulation Management Component to other Components using a shared Memory Location. A “Management Portal” argument is present on all Components, and there should be one Management Portal in existence at the top of the Simulation that is shared by all Components. The Simulation Management Component writes to this portal, whereas all other Components read from it.

Although Probes can be placed anywhere, there are two special cases that relate to the Transport Layer and Network Layer. The TCP and Queue modules in these, respectively, are constructed in ‘C’, and therefore inaccessible to existing BONeS probes. Therefore, special “Transport -- TCP” and “Network -- Queue” probes have been created that have an argument specifying the type of data to retrieved. These arguments are covered in the respective implementation details.

Finally, the Management File is at the heart of the Simulation. It is read and translated into Information Elements that are passed to appropriate Modules within Addressed Components. Note that for the case of LANs, which involve aggregations of Components, the Address must correspond to a constituent Component (i.e. Host, Traffic, Router or Link). The File is constructed with ASCII numbers; a program to convert between friendlier strings and numbers was to be constructed, but it did not eventuate. The following paragraphs and tables detail the format of this File:

3.5.1. Management File Format

The File is a standard text file; it consists of a number of lines. Each line corresponds to a particular command that is composed of a number of fields. Each field is separated by a number of spaces.

The first field provides the Time at which the command must execute. This is a number of seconds relative to the start of the Simulation. It is encoded as a REAL number, allowing for fractional seconds.

The second field provides the Destination Address for the command. This corresponds to the Address argument for a particular Component in the Simulation. It is encoded as an INTEGER number.

The third field provides the Destination Module for the command. This corresponds to a particular type of Module within the given Component that the command is addressed to. Table 1-3.40 illustrates the values that this field can take. It is encoded as an INTEGER number.

Destination Module	*Value*
Datalink Layer	00
Network Layer	01
Transport Layer	02
Network-Adaption Layer	03
Transport-Adaption Layer	04
Generator	05
Routing-Module	06

Table 1-3.40. Management File: Destination Modules

The fourth field provides the type of Command. The Command Types are particular to each given Destination Module. Table 1-3.41 illustrates the values that this field can take. It is encoded as an INTEGER number.

Destination Module	*Command Type*	*Value*
Datalink Layer	Set State	00
Network Layer	-	-
Transport Layer	Set Parameters	00
Network-Adaption Layer	Set Address List	00
Transport-Adaption Layer	Connect Session	00
Transport-Adaption Layer	Disconnect Session	01
Generator	Setup FTP Generator	00
Generator	Setup Telnet Generator	01
Generator	Setup Statistical Generator	02
Routing-Module	Set Route Entry	00

Table 1-3.41. Management File: Command Types

For each Command Type, there are arguments particular to that Command Type. The following tables provide the content details for each Command Type.

3.5.1.1. Datalink Layer -- Set State

The operational state of the Datalink Layer can be set to active or inactive. When the state is set to inactive, the Datalink will not transport Messages between its two connected peers, it will whist active. In both cases, it indicates to its peer Network Layer the state that it has entered. The state is supplied as an argument.

Number	*Content*
1	An INTEGER number providing indicating what state the Datalink Layer should be in, having the following values: 00: Inactive 01: Active

3.5.1.2. Transport Layer -- Set Parameters

When the Transport Layer establishes a connection with a peer, it does not carry out any TCP handshaking (this is a model, not an actual implementation of TCP), therefore some means to specify the Initial Sequence Number (ISN) is required, so that both peers can communicate correctly. This is not set at a fixed value, but it will default to one, as crossover connections are still potentially possible in our environment. The Initial Sequence Number (ISN) is supplied as an argument.

Number	*Content*
1	An INTEGER number, of any value, corresponding to the Initial Sequence Number (ISN) that should be used.

3.5.1.3. Network-Adaption Layer -- Set Address List

The Network-Adaption Layer encapsulates units of data and transports them using Network Layer Messages, in doing so, it must provide addressing information for these Messages. A List of Addresses can be supplied, and the Network-Adaption Layer will select one at random for each Message that it creates, naturally, if only one Address is supplied in the List, then it will be used at all times. The encoding consists of specifying the number of entries in the Address List, and then each Address in succession.

Number	*Content*
1	An INTEGER number, of a value between 1 and 32 inclusive, corresponding to the number of Addresses that are provided in the list.
2	An INTEGER number, corresponding to the first Address in the Address List.
3	An INTEGER number, corresponding to the second Address in the Address List.
...	
n	An INTEGER number, corresponding to the nth Address in the Address List.

3.5.1.4. Transport-Adaption Layer -- Connect Session

The Transport-Adaption Layer uses the Transport Layer, and in particular may request the Connection of a Transport Session. When doing so, it must indicate a peer Address for the session. The Address is supplied as an Argument.

Number	*Content*
1	An INTEGER number, corresponding to the Destination Address for the Transport Session.

3.5.1.5. Transport-Adaption Layer -- Disconnect Session

The Transport-Adaption Layer uses the Transport Layer, and in particular may request the Disconnection of a Transport Session. It does not need to convey any additional information, so there are no arguments in this case.

Number	*Content*

3.5.1.6. Generator -- Setup FTP Generator

The Generator can provide data corresponding to an FTP profile. In addition, it operates according to a set of filter parameters that allow for limitations to be set on the length of time that data is generated for, the number of bytes that is generated in total, and the number of units of data that is generated. These three common filter parameters are supplied as arguments.

Number	*Content*
1	A REAL number, indicating the maximum length of time that the Generator should continue providing data for. The value is in seconds, and may be fractional. This does not apply for FTP data, as there is only one item created.
2	An INTEGER number, indicating the maximum number of bytes that the Generator should provide.
3	An INTEGER number, indicating the maximum number of units of data that the Generator should provide. This does not apply for FTP data, as there is only one item created.

3.5.1.7. Generator -- Setup Telnet Generator

The Generator can provide data corresponding to an Telnet profile. In addition, it operates according to a set of filter parameters that allow for limitations to be set on the length of time that data is generated for, the number of bytes that is generated in total, and the number of units of data that is generated. These three common filter parameters are supplied as arguments.

Number	*Content*
1	A REAL number, indicating the maximum length of time that the Generator should continue providing data for. The value is in seconds, and may be fractional.
2	An INTEGER number, indicating the maximum number of bytes that the Generator should provide.
3	An INTEGER number, indicating the maximum number of units of data that the Generator should provide.

3.5.1.8. Generator -- Setup Statistical Generator

The Generator can provide data corresponding to a statistical distribution. In addition, it operates according to a set of filter parameters that allow for limitations to be set on the length of time that data is generated for, the number of bytes that is generated in total, and the number of units of data that is generated. These three common filter parameters are supplied as arguments. There are a number of potential statistical distributions, each of which has defined arguments. Two sets of distributions are supplied, data is generated at intervals corresponding to the Time characteristic, and the amount of data generated corresponds to the Space characteristic.

Number	*Content*
1	A REAL number, indicating the maximum length of time that the Generator should continue providing data for. The value is in seconds, and may be fractional.
2	An INTEGER number, indicating the maximum number of bytes that the Generator should provide.
3	An INTEGER number, indicating the maximum number of units of data that the Generator should provide.
4 ... a	An ENCODING of parameters for a particular Statistical Distribution, corresponding to the Time characteristic needing to be generated.
a+1 ... b	An ENCODING of parameters for a particular Statistical Distribution, corresponding to the Space characteristic needing to be generated.

3.5.1.8.1. Statistical Parameter Encoding -- Constant

The Statistical Parameter encoding includes an identifier for the type of distribution, and a single Constant value to characterise a Constant Statistical Distribution.

Number	*Content*
1	An INTEGER number, indicating the particular type of Statistical Parameter, which has the value: 00: Constant Distribution
2	A REAL number, indicating the constant value.

3.5.1.8.2. Statistical Parameter Encoding -- Uniform

The Statistical Parameter encoding includes an identifier for the type of distribution, and Lower and Upper bounds to characterise a Uniform Statistical Distribution.

Number	*Content*
1	An INTEGER number, indicating the particular type of Statistical Parameter, which has the value: 01: Uniform Distribution
2	A REAL number, corresponding to the Minimum value in the Uniform Distribution.
3	A REAL number, corresponding to the Maximum value in the Uniform Distribution.

3.5.1.8.3. Statistical Parameter Encoding -- Normal

The Statistical Parameter encoding includes an identifier for the type of distribution, and Mean and Variance values to characterise a Normal Statistical Distribution.

Number	*Content*
1	An INTEGER number, indicating the particular type of Statistical Parameter, which has the value: 02: Normal Distribution
2	A REAL number, corresponding to the Mean value in the Normal Distribution.
3	A REAL number, corresponding to the Variance value in the Normal Distribution.

3.5.1.8.4. Statistical Parameter Encoding -- Exponential

The Statistical Parameter encoding includes an identifier for the type of distribution, and a Mean value to characterise an Exponential Statistical Distribution.

Number	*Content*
1	An INTEGER number, indicating the particular type of Statistical Parameter, which has the value: 03: Exponential Distribution
2	A REAL number, corresponding to the Mean value in the Exponential Distribution.

3.5.1.8.5. Statistical Parameter Encoding -- Poisson

The Statistical Parameter encoding includes an identifier for the type of distribution, and a Lambda value to characterise a Poisson Statistical Distribution.

Number	*Content*
1	An INTEGER number, indicating the particular type of Statistical Parameter, which has the value: 04: Poisson Distribution
2	A REAL number, corresponding to the Lambda value in the Poisson Distribution.

3.5.1.9. Routing-Module -- Set Route Entry

The Routing-Module maintains a table of Route entries that are used to switch packets between connected Links using Interfaces with those Links. An entry can be placed into this table, containing an Address and the Interface used to reach that Address, at a given Cost. An argument is used to provide each value.

Number	*Content*
1	An INTEGER number, indicating the Address for which this Routing Entry is for.
2	An INTEGER number, indicating the Interface on the particular Routing-Module for which the Route corresponds to. This has a value between 1 and 5 inclusive.
3	A REAL number, indicating the Cost associated with the Route. A value of 0.0 indicates that there is no Route.

3.5.2. Management File Example

The following illustrates a simple example of the Management File. Consider the requirements of needing to set up a Router between two Hosts, and then establish a TCP conversation between these two Hosts. Let the first Host be denoted by *A* (1), and the second Host by *B* (2), and the Router by *C* (3). The Route costs are irrelevant in this context.

The conversation will consist of Telnet traffic, only in one direction; from *A* to *B*. The TCP conversation starts at Time 1 (let the system initialise) and proceeds for 10 seconds, after which the simulation terminates at Time 12. Table 1-3.42 outlines the exact commands and file contents.

Time	*Addr*	*Command*	*Description*	*File Contents*
0	C	Set Route Entry (A,1,1)	Interface 1 reaches *A* at Cost 1	00 03 06 00 01 01 01
0	C	Set Route Entry (B,2,1)	Interface 2 reaches *B* at Cost 1	00 03 06 00 02 02 01
0	A	Set Parameter (1234)	TCP use an ISN of 1234	00 01 02 00 1234
0	B	Set Parameter (1234)	TCP use an ISN of 1234	00 02 02 00 1234
0	A	Connect Session (B)	TCP Connect to B	00 01 04 00 02
0	B	Connect Session (A)	TCP Connect to A	00 02 04 00 01
1	A	Setup Telnet Generator (10,0,0)	Telnet data for 10 seconds, no byte or unit restrictions	01 01 05 01 10 00 00
12	-	-	End of Simulation	12

Table 1-3.42. Management File: Example

4. Testing

4.1. Overview

The BONeS Modules constructed through the implementation need to be verified to ensure that they function according to their design. Modules that do not function correctly will result in the collection of invalid data during the execution of simulations, and therefore invalid analysis and conclusions.

To achieve this, two steps where considered. The first step consists of per Module testing, in which each Module is individually tested by itself. The general strategy is to construct a simulation in which the Module is stimulated by inputs, and outputs are logged to a file. The contents of the file are then examined to determine whether correct behaviour has occurred.

The second step consists of an execution of more complex simulations, for the express purpose of gaining results that can be correlated with previous work and theoretical expectations. The first two simulations designed for examination, in Part 2, have verification and validation as their express objectives.

Due to the BONeS software not being available for use, this testing could not be carried out. However, the 'C' modules, with the exception of the TCP implementation where tested through their development, which is why the implementation of these modules is quite modular in nature (crisp boundaries and interfaces are more amenable to testing).

The following summary illustrates the basic strategy employed to test each Module; the tests do not cover every single aspect of behaviour, as such tests would be too involved and any other anomalies can be picked up during the simulations.

4.2. Summary

4.2.1. Datalink Layer

For the correct behaviour of the Datalink Layer, the following points need to be verified:

- The correct delay is introduced, according to the Bandwidth and Propagation characteristics set for the Datalink Layer.
- When the Datalink Layer is in the inactive state, it does not pass Messages are.
- When Messages are sent, while the Datalink Layer is in the active state, Messages are passed through.

The following steps are followed to carry out testing for these points:

1. A BONeS simulation is constructed with the following content:
    - A single Datalink Layer Module.
    - A loop started by an "Init" Module that generates a Datalink Data Request Message every 1 second. The lengths of the Messages are set to random

    values (a uniform distribution will suffice). These are sent to the Datalink Layer input port.

    - A "One Pulse" Module, set to trigger at 10 seconds, which causes a Management Messages to be generated and written into the Management Portal. The Message indicates that the state of the Datalink Layer is inactive..
    - A "One Pulse" Module, set to trigger at 20 seconds, which causes a Management Messages to be generated and written into the Management Portal. The Message indicates that the state of the Datalink Layer is active.
    - "Textual Description Probes" are placed on all input and output ports of the Datalink Layer.
    - The remaining Datalink Layer parameters are set to conservative values.

2. The simulation is run for 30 seconds, during which time all inputs and outputs are written to the log file.
3. The contents of the log file are examined to ensure that
    - The delay incurred by each Message is correct according to the configured Bandwidth, Propagation Delay and the particular Message's length.
    - No Messages are output from the Datalink Layer whilst it is in the inactive state.

4.2.2. Network Layer

For the correct behaviour of the Network Layer, the following points need to be verified:

- Messages are enqueued.
- Messages are rejected upon insertion into a full queue.
- Messages are released when instructed.
- The various queue behavioural disciplines function correctly.

The following steps are followed to carry out testing for these points:

1. A BONeS simulation is constructed with the following content:
    - A single Network Layer Module.
    - An "Init" Module set to trigger an infinite loop. The loop iterates with period 1 second, and generates a Network Layer Data Request Message of arbitrary length, with sequentially increasing destination address. These are sent to the Network Layer input port.
    - An "Init" Module, which generates a Datalink Connect Indication Message to inform the Network Layer that the Datalink Layer is active. This is sent to the Datalink Layer input port.
    - The queue length is set to 5 packets.

    - A “One Pulse” Module, set to execute at 10 seconds, that triggers a loop. The loop iterates with period 0.5 seconds, and generates Datalink Layer Status Indication of Flow Control Released Messages. These are sent to the Datalink Layer input port.
    - “Textual Description Probes” are placed on all input and output ports of the Network Layer.
    - The remaining Network Layer parameters are set to conservative values.
2. The simulation is run for 30 seconds, during which time all inputs and outputs are written to the log file.
3. The contents of the log file are examined to ensure that
    - The Network Status Indication Messages with Load Factor Information Elements indicate that the queue size is growing.
    - The queue overflows when it becomes full, and therefore results in the removal of a Message.
    - The selection of a Message to destroy (when overflow occurs), and the Messages output are consistent with the particular queue discipline in place.

4.2.3. Transport Layer

The testing for the Transport Layer is left until the simulations. All other Modules can be verified prior to simulation, so the simulation serves to carry out this verification (and validation).

4.2.4. Network-Adaption Layer

For the correct behaviour of the Network-Adaption Layer, the following points need to be verified:

- The Address-List can be set and its content is used in the generation of Messages.
- Input of a single item of data results in the output of a Network Layer Data Request with appropriate fields set.

The following steps are followed to carry out testing for these points:

1. A BONeS simulation is constructed with the following content:
    - A single Network-Adaption Layer Module.
    - An “Init” Module, to trigger the generation of a Management Message, which is written to the Management Portal. The Message contains an Set Address-List Information Element with random Addresses.
    - An “Init” Module, to trigger an infinite loop. The loop iterates with period 1 second, and generates random integers. These are sent to the upper layer input port.
    - A “One Pulse” Module set to execute at 10 seconds, which triggers the generation of a Network Layer Connect Indication. This is sent to the Network Layer input port.

    - “Textual Description Probes” are placed on all input and output ports of the Network-Adaption Layer.
2. The simulation is run for 20 seconds, during which time all inputs and outputs are written to the log file.
3. The contents of the log file are examined to ensure that
    - No Messages are generated for the first 10 seconds while the Network Layer is (apparently) not active.
    - Messages are generated between 10 and 20 seconds that have correct length and select a random Address from the Address-List.

4.2.5. Transport-Adaption Layer

For the correct behaviour of the Transport-Adaption Layer, the following points need to be verified:

- Management Connect and Disconnect operations result in Transport Connect and Disconnect Messages.
- Input of a single item of data results in the output of a Transport Layer Data Request with appropriate fields set.

The following steps are followed to carry out testing for these points:

1. A BONeS simulation is constructed with the following content:
    - A single Transport-Adaption Layer Module.
    - An “Init” Module, to trigger an infinite loop. The loop iterates with period 1 second, and generates random integers. These are sent to the upper layer input port.
    - A “One Pulse” Module set to execute at 10 seconds, which triggers the generation of a Management Message, which is written to the Management Portal. The Message contains a Session Connect Information Element with a random Address.
    - A “One Pulse” Module set to execute at 20 seconds, which triggers the generation of a Management Message, which is written to the Management Portal. The Message contains a Session Disconnect Information Element.
    - “Textual Description Probes” are placed on all input and output ports of the Transport-Adaption Layer.
2. The simulation is run for 30 seconds, during which time all inputs and outputs are written to the log file.
3. The contents of the log file are examined to ensure that
    - Messages are generated that have correct lengths.
    - Connect and Disconnect Messages are generated and have correct contents.

4.2.6. Routing-Module

For the correct behaviour of the Routing-Module, the following points need to be verified:

- Routing for a single case works.
- Routing for multiple cases works.

The following steps are followed to carry out testing for these points:

1. A BONeS simulation is constructed with the following content:
    - A single Routing-Module Layer Module.
    - An "Init" Module, to trigger an infinite loop. The loop iterates with period 1 second, and generates Network Data Indication Messages. These are set with random Addresses between 1 and 5 inclusive, and to random input ports on the Routing-Module.
    - An "Init" Module that triggers the generation of Management Messages, which are written to the Management Portal. The Messages contain Routing Entries for Addresses 1 to 5 inclusive.
    - A "One Pulse" Module set to execute at 10 seconds, which triggers the generation of Management Messages, which are written to the Management Portal. The Messages contain Routing Entries for Addresses 1 to 5 inclusive. This sets up multiple paths.
    - "Textual Description Probes" are placed on all input and output ports of the Routing-Module.
2. The simulation is run for 45 seconds, during which time all inputs and outputs are written to the log file.
3. The contents of the log file are examined to ensure that
    - The initial routes work correctly and Messages are appropriately placed.
    - The costing mechanism for multiple routes works correctly.

4.2.7. Generator

For the correct behaviour of the Generator, the following points need to be verified:

- All different types of Generators can be Setup; i.e. Telnet, FTP and Statistical
- Filter parameters will correct limit the creation of data.

The following steps are followed to carry out testing for these points:

1. A BONeS simulation is constructed with the following content:
    - A single Generator Module.
    - A "One Pulse" Module, set to execute at 0 seconds, that triggers the generation of a Management Message which is written to the Management Portal. The Message contains Setup Generator FTP request, with a time constraint of 5 seconds.

    - A “One Pulse” Module set to execute at 10 seconds, which triggers the generation of a Management Message, which is written to the Management Portal. The Message contains Setup Generator Telnet request, with a time constraint of 5 seconds.
    - A “One Pulse” Module set to execute at 20 seconds, which triggers the generation of a Management Message, which is written to the Management Portal. The Message contains Setup Generator Statistical request, with a constant amount of data (1 byte) being output periodically (1 second). A byte constraint of 5 is used.
    - A “One Pulse” Module set to execute at 30 seconds, which triggers the generation of a Management Message, which is written to the Management Portal. The Message contains Setup Generator Statistical request, with a constant amount of data (1 byte) being output periodically (1 second). A unit constraint of 5 is used.
    - “Textual Description Probes” are placed on all input and output ports of the Generator.
2. The simulation is run for 40 seconds, during which time all inputs and outputs are written to the log file.
3. The contents of the log file are examined to ensure that
    - Appropriate data is generated at the particular times.
    - The filter parameters terminate the generator at the requested time.

4.2.8. Management

For the correct behaviour of the Management Module, the following points need to be verified:

- Each Command Type can be read and parsed.
- Time and Addressing information is correctly used.

The following steps are followed to carry out testing for these points:

1. A BONeS simulation is constructed with the following content:
    - A single Management Module.
    - Five Management Portals, with addresses 1, 2, 3, 4 and 5.
    - A Management File with each different Command Type combination, with a command every 1 second using a random address between 1 and 5 inclusive.
    - “Textual Description Probes” are placed on all input and output ports of the Management Module.
2. The simulation is run until termination, during which time all inputs and outputs are written to the log file.
3. The contents of the log file are examined to ensure that

- All commands were parsed and Information Elements were created with the correct content.
- Management Messages were only received through Management Portals corresponding to the Address associated with the command.

PART 2. CONSTRUCTION, EXECUTION AND ANALYSIS OF SIMULATIONS

1. Introduction

The central objectives in this thesis are concerned with the behaviour of congestion control in Wide-Area Networks (WANs) as they apply to the Transmission Control Protocol (TCP). The examination of this behaviour is achieved through the construction, execution and analysis of simulations using the integrated BONeS environment. The environment provides an ability to build and execute simulations then collect, post-process and display simulation results without the need to manipulate raw data sets.

The approach taken here consists of a stepwise process, starting with a basic outline of the problem, and the abstract objectives to be reached in relation to that problem. Following this is a more detailed discussion relating to the problem that covers any related work on the issue. Subsequently, an approach is developed: this consists of a high level outline of the model to be used, and the simulation to take place.

Next, the abstract simulation is transformed into a BONeS simulation. This consists of a model, having a particular topology and static parameters. The static model is promoted to a dynamic model by the addition of a runtime management script and observational probes. It is then executed, possibly with a number of variations, and the results, from the probes, are post-processed ready for interpretation and analysis.

Following the design of the simulation, it is possible to outline the expectations. This is an important aspect, as it is unwise to carry out simulations without prior expectations, however abstract they may be. The final stages in the process consist of executing, analysing and drawing conclusions on the simulation.

Unfortunately, due to the unforseen circumstances surrounding this work, the final stages of execution, analysis and conclusions could not be reached. Some conclusions are presented based upon the expectations, however in some cases it has been difficult to determine the expectations, as the behaviour is not well known.

2. Simulation Strategies

In general, all the simulations attempt to use models that are representative of practical environments. Naturally, by virtue of the simplified nature of the models and the simulation environment, this is still very much theoretical in nature.

A general problem related to simulation models is that of parameter sensitivity. This is a situation where parameters used in the model, or simulation, bias results, and minor variations cause significant differences in the results obtained. All simulations are designed to run multiple times with iteration on the “Global Seed” parameter, as recommended by BONeS. As a general principle, however, the results of all simulations are subject to analysis to ensure that they are free from recognised defects.

The attempt is to build all models and simulations upon a well-founded basis, by ensuring that models are legitimate in their representation and parameters are realistic. Based on this, and known theory and investigation into related work, it is possible to develop expectations related to the simulations and their outcomes. This is a process fundamental to all work involving experimentation. It is given careful treatment here.

3. Simulation Scenarios

3.1. Single TCP Conversation

3.1.1. Problem and Objectives

The case of a single TCP conversation is intended to look at the very basic nature of TCP and congestion control.

Primarily, it exists as an exercise in verification and validation. The results gained from this simple, well-understood and well-examined scenario are compared against theoretical expectations and prior work. This process gives a high degree of assurance that the BONeS Modules are functioning correctly and acting as representative models. Therefore, *the first objective is to carry out verification and validation.*

This simple, and in some respects idealistic, scenario is also a perfect instrument for demonstrating the basic fundamental behaviour of TCP congestion control. By such a demonstration, a more precise understanding is developed prior to subsequent scenarios that assume and build upon this understanding. Therefore, *the second objective is to explain the basic fundamental behaviour of TCP congestion avoidance and control.*

3.1.2. Discussion and Related Work (NOT FINISHED)

Single TCP conversations have been well studied, not only in the context of the BSD 4.4 congestion control used here [ref], but for other congestion control measures. some of this work is particularly focused upon the nature of the congestion control, but others have a primary focus other than the congestion control [ref].

a work that is of particular interest is [ref], as it tends to look directly at bsd 4.4, which is precisely what is being used here. it should be noted that there are various incarnations of tcp congestion control, such as “tahoe” and “reno”. “tahoe” basically refers to <x>, whereas “reno” <x> [ref]. The differences tend to cloud the central issue.

using this approach for verification and validation has been carried out before [ref], so this is not a problem, and although we most likely will not get exact characterstics, we will tend to see the basic behaviour present. if the behaviour is not equivalent, then we can conclude that our model and simulation is incorrect.

when considering expectations, we can best carry out an examination by a walkthough of the conversations lifecycle, with reference to expected graphs to be produced.

3.1.3. Approach

The approach consists of identifying the model, simulation and observations that are required to obtain the objectives.

Model

The model consists of a point-to-point TCP conversation between two Hosts. The conversation occurs through an intermediate Link, which provides the primary resource constraint in the network. Each Host also has a network queue, which provides the secondary resource constraint. The model is specifically simple, it does not employ routers.

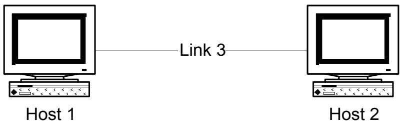


Figure 2-3.1. Simulation Model: Single TCP Conversation LAN

The model has parameters, some of which change during the execution of the simulations. The Link's Bandwidth and Propagation Delay do not change, they are set to realistic values of 64kbps and 20ms respectively, intended to represent a typical WAN situation. The queue disciplines are set to Drop Tail, as they are not of concern in this particular scenario.

Basic Simulation

The basic simulation consists of a conversation between Host 1 and Host 2, through Link 3. The conversation starts at time 0, and proceeds for enough time to capture an expected transient and state steady response. 30 seconds should be sufficient for this purpose. The conversation carries traffic (a transfer of a large unit of data) from Host 1 to Host 2, the return traffic only consists of acknowledgements.

Variations

Only one variation is considered: alteration of the queue length. Through this, it is possible to see the effects of altered Round Trip Time (RTT) and the general effects related to smaller or larger queue lengths. The queue length is iterated for values between 1 and 24 (packets).

Observations

The concern is with the nature of the TCP congestion control mechanisms, so observations are made of the TCP transmitter in Host 1. TCP congestion control mechanisms are transmitter based, so there are no items of interest in Host 2. To correlate the TCP congestion control with network conditions, observations are made of the queue length and the link utilisation. The correlations are important in illustrating the operation of TCP's congestion control mechanisms.

3.1.4. BONeS Simulation Design

Transfer from an abstract approach into a simulation first requires the construction a BONeS simulation module. Probes are then placed into this module to capture data during the simulation, noting that for all runs the same probe configuration is used (this is done for simplicity). The operation of the Basic Simulation, with details about Parameters and execution script, is given, after which the modifications are described for each subsequent iteration.

Every simulation is run with iteration of the "Global Seed" Parameter, at least three times. This particular aspect is not explicitly outlined because it is carried out so that visual observation can be made to ensure that results are correct. It is fortunate that the automated capability of BONeS allows for this to be carried out quickly and effortlessly.

3.1.4.1. Topology

The approach is translated into an actual BONeS simulation first through the construction of a simulation Module using the components developed on Part 1 of this thesis. The parameters relevant to the simulation are visible in the figure.

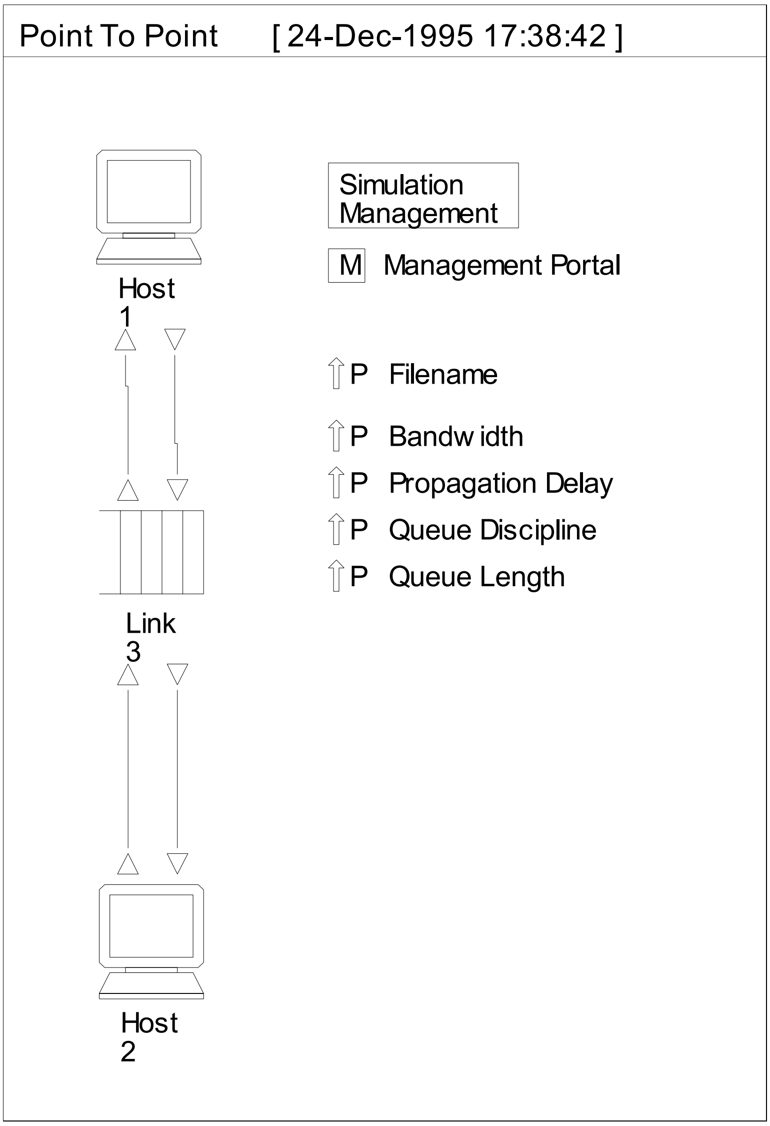


Figure 2-3.2. Simulation Topology: Single TCP Conversation LAN

3.1.4.2. Post Processing and Probe Placement

To construct information used in the analysis, Probes can be placed into the simulation using the BONeS Simulation Manager; once placed, they are then used in the Post Processor to generate graphs. The approach taken here is to first identify the particular graphs that indicate critical information for analysis, and then to determine which Probes must be placed, and where they must be placed.

3.1.4.2.1. Basic Simulation

For the basic simulation, the graphs illustrate the lifecycle activity in the host and the network.

TCP Window Information	
For	Host 1
Purpose	To show the detailed attributes of the TCP congestion control algorithm, as it alters during the course of the simulation. In addition, events that are correlated with TCP congestion control activity are also captured.
X Axis	(Seconds): Time
Y Axis	(Bytes): Congestion Window, Slow Start Threshold, Unacknowledged Data (No Units): Retransmission Events, Timer Expiries
Probes	TCP Probes are used, and they are placed into Host 1's Transport Layer.

TCP Computed and Actual Round Trip Time (RTT) Information	
For	Host 1
Purpose	The RTT plays an important role in TCP congestion control. However, as it is estimated, observations of the actual RTT should also be made.
X Axis	(Seconds): Time
Y Axis	(Milliseconds): RTT Value, RTT Value +RTT Variance, RTT Value - RTT Variance, Actual RTT
Probes	TCP Probes are placed into Host 1's Transport Layer. The Actual RTT is obtained by placing a probe into Host 1's Transport Layer to extract the timing information from a received acknowledgement.

Queue Information	
For	Host 1
Purpose	The queue drops packets, and affects the RTT for packets. Its behaviour can be correlated with that of TCP congestion control.
X Axis	(Seconds): Time
Y Axis	(Integer Value): Queue Length, Queue Drops
Probes	Queue Probes are placed into Host 1's Network Layer.

Transport Layer Data Transmission	
For	Host 1
Purpose	The qualitative information about a conversation is related to its throughput and retransmission levels. The number of transmitted and retransmitted bytes is also affected, and can be correlated with, TCP congestion control activity.
X Axis	(Seconds): Time
Y Axis	(Kilobytes): KB Transmitted, KB Retransmitted
Probes	TCP Probes are placed into Host 1's Transport Layer.

Transport Layer Data Transmission (95% confidence level)	
For	Host 1
Purpose	For greater confidence in the simulation results, a confidence plot using different initial random seeds is used. The information best used on a confidence plot is the throughput and retransmit levels, as the assumption is that they are relevant equivalent for a given scenario. Window and Queue information is more highly variant, and subject to phase differences.
X Axis	(Seconds): Time
Y Axis	(Kilobytes): KB Transmitted, KB Retransmitted
Probes	TCP Probes are placed into Host 1’s Transport Layer.

Link Utilisation	
For	Link 3
Purpose	Because of retransmission timeouts and other events, the link may not always be fully utilised, where under ideal conditions it should always be.
X Axis	(Seconds): Time
Y Axis	(Percentage): Utilisation
Probes	Probes are placed into Link 3. They capture the sum of all packet lengths passed through the link over the total capacity made available by that link according to the length of time in the simulation.

3.1.4.2.2. Queue Length Iteration

When the queue length is iterated, the same graphs are constructed as in the Basic Simulation. In addition, the variation in specific items as a function of the Queue Length becomes of interest.

Throughput versus Queue Length	
For	Host 1
Purpose	The relationship between Queue Length and Throughput tends to indicate a “good” queue length, and the effects of queuing in general (in a first or second order manner).
X Axis	(Integer Value): Queue Length
Y Axis	(Kilobytes per second): Throughput
Probes	The Probes used are those from the Basic Simulation. The graph is constructed by taking the total number of bytes transmitted for the conversation over the time of the conversation, for each simulation run.

Average RTT versus Queue Length	
For	Host 1
Purpose	As Queue Length is increased, the RTT should be noticeably different both in average value and variance.
X Axis	(Integer Value): Queue Length
Y Axis	(Milliseconds): Average RTT Value, Average RTT Variance, Average actual RTT
Probes	The Probes used are those from the Basic Simulation. The graph is constructed by taking the average RTT for the conversation for each simulation run.

Retranmission Ratio versus Queue Length	
For	Host 1
Purpose	Queue Lengths and Retransmission Ratios may be correlated. The retransmission ratio is determined by taking the total number of retransmitted bytes for a conversation and dividing by the total number of transmitted bytes.
X Axis	(Integer Value): Queue Length
Y Axis	(Integer): Retransmission Ratio
Probes	The Probes used are those from the Basic Simulation. The graph is constructed by computing the retransmission ratio for the conversation for each simulation run.

3.1.4.3. Execution: Basic Simulation

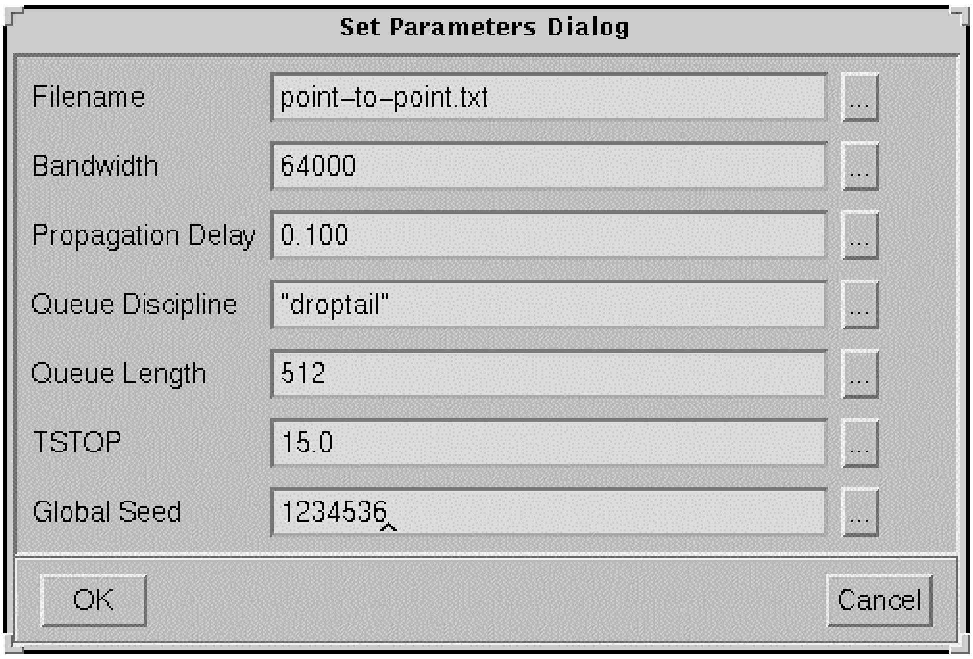
In the basic simulation, the Parameters must be configured using the BONeS Set Parameters Dialog. One such parameter is the Management Script. There is no iteration in the basic simulation.

3.1.4.3.1. Parameters

The parameters correspond to the values discussed in the Approach.

Parameter	*Value*	*Description*
Filename	point-to-point.txt	The file contains the Management Script.
Bandwidth	64kbps	Models an ISDN B Channel.
Propagation Delay	20ms	Models a potential interstate delay.
Queue Discipline	Drop Tail	The value here is irrelevant.
Queue Length	4	Too large a value will result in excessive delay, whereas too low a value will prohibit fast retransmit from occurring.

The “Set Parameters Dialog” in BONeS actually looks like (although, the queue length and the simulation is incorrect).



Set Parameters Dialog

Filename	point-to-point.txt
Bandwidth	64000
Propagation Delay	0.100
Queue Discipline	"droptail"
Queue Length	512
TSTOP	15.0
Global Seed	1234536

Figure 2-3.3. Simulation Config: Single TCP Conversation LAN

3.1.4.3.2. Management Script

The Management Script is broken up into a number of steps according to the outline given in the Approach.

Step 1: Initial Configuration at Time 0

None -- There is no initial configuration to perform.

Step 2: Establishment of TCP conversation between Host 1 and Host 2 at Time 0

Set Initial Sequence Numbers for Host 1 and Host 2 -- The initial sequence numbers are an arbitrary value, they are not important other than the fact that both have the same value.

```
0 ⇒ Host 1 : Set Parameters (ISN: 12345678)
0 ⇒ Host 2 : Set Parameters (ISN: 12345678)
```

Request Host 1 to Connect Session to Host 2, and Host 2 to Connect Session to Host 1 -- The TCP conversations now enter the ESTABLISHED state, although they do not communicate as no data is available.

```
0 ⇒ Host 1 : Connect Session (Addr: Host 2)
0 ⇒ Host 2 : Connect Session (Addr: Host 1)
```

Instruct the Generator on Host 1 to produce a single Constant unit of data -- The Generator supplies a unit of data to the TCP conversation, which proceeds to transfer this unit of data to Host 2. There are no filter constraints.

```
0 ⇒ Host 1 : Setup Statistical Generator (Time: 0, Bytes: 0, Count: 0,
Time   (Type:   CONSTANT,   Value:   1),   Space(Type:   CONSTANT,   Value:
10000000))
```

Step 3: Terminate the simulation at time 30

Stop -- The absence of any more commands is an indication to stop.

```
30 ⇒ :
```

The Management script is constructed by translating these pseudo operations using the information provided in Part 1. This is not provided here, as it is cryptic and pointless.

3.1.4.4. Execution: Queue Length Iteration Simulation

The same Parameters and Management script are used as in the Basic Simulation, however for the Queue Length Parameter a BONeS iteration dialog is selected. This dialog is instructed to step through the Queue Length from values 1 to 24 inclusive.

3.1.5. Expectations (NOT FINISHED)

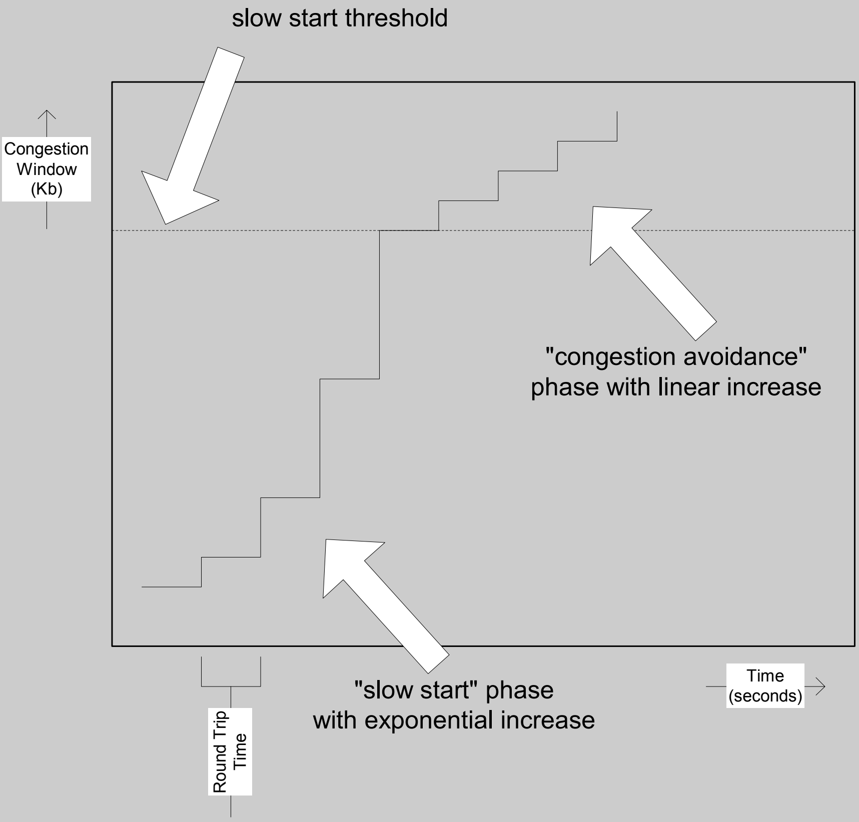
When the simulation commences, the TCP conversation begins operation. Initially, the conversation sets the congestion window to the size of one segment (512 bytes), and it sets the slow start threshold to the maximum window size (64K). This is evidenced in the following code:

> <put code here that sets up>

TCP then transmits a single segment, and after subsequent delay receives an acknowledgement back through the network from the receiver. When an acknowledgment is received, the congestion window increases. Initially, the congestion window is lower than the slow start threshold, so “slow start” is performed. This causes the transmitter to increase the congestion window exponentially. The following code is executed to achieve this:

> <put code here that does exp increase>.

At first glance, the code seems to carry out a linear increase, but this is misleading. Consider that with a congestion window size of one (segment), the transmitter can only generate one segment into the network. When the acknowledgement is received for this one segment, it increases the window by one, and it can transmit two segments. When *each* of these two acknowledgements are received, it increases the window by one, and therefore can transmit four segments. When four acknowledgements are received, the window is increased by one for each acknowledgement, and thence becomes eight, ... and so on. This “slow start” phase allows the transmitter to rapidly increase the window until it reaches the “safe” slow start threshold (Jacobson, 1988). Figure X illustrates the behaviour of the window during this phase.


slow start threshold
Congestion Window (Kb)
"congestion avoidance" phase with linear increase
"slow start" phase with exponential increase
Time (seconds)
Round Trip Time

Initially, with the slow start threshold set to 64K, the conversation will tend to always suffer congestion before it reaches the slow start threshold, but consider for the moment if the slow start threshold were less than 64K.

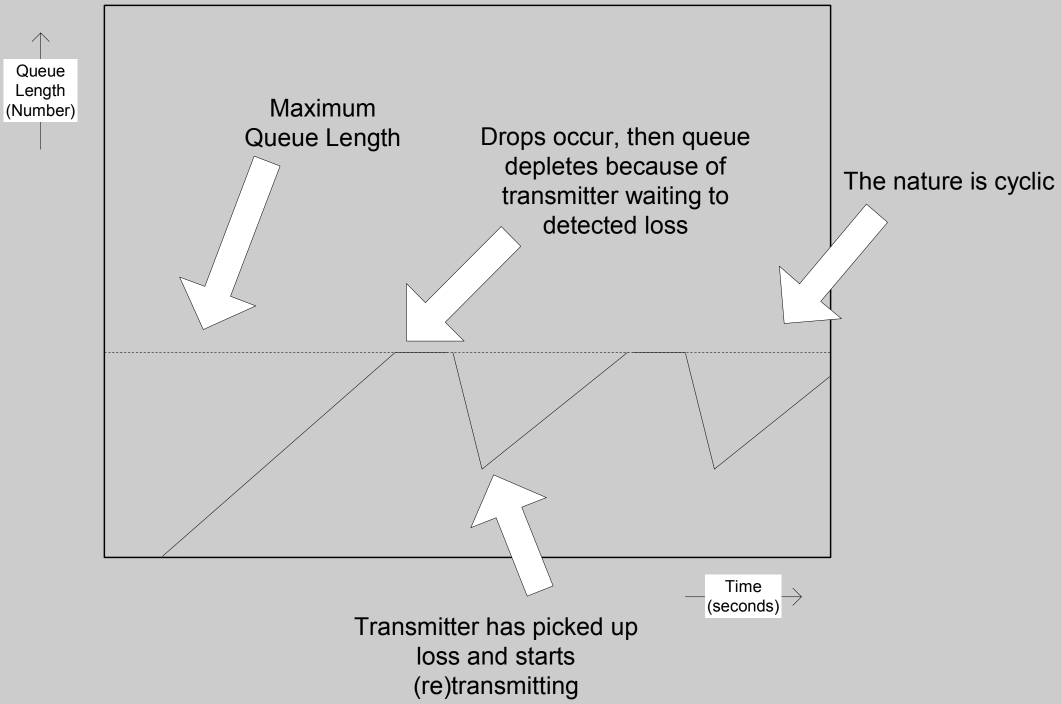
If the transmitter does not experience congestion, then at some stage the congestion window will be advanced beyond the slow start threshold. From this point, the transmitter changes over to the "congestion avoidance" phase (Jacobson, 1988) and the window is increased linearly, according to the following code:

<put code in here that does linear increase>

As before, this code may seem intuitively incorrect, but first impressions are misleading. By increasing the window by the inverse of the window, for all acknowledgements in the window, then the sum of all the inverse window values will equal one. Hence, the window will increase by one segment. The "congestion avoidance" phase is used by the transmitter to slowly probe the network in an attempt to reach the network's operating point (Jacobson, 1988). Figure X illustrates the increase in the congestion window after it has reached the slow start threshold.

In the results obtained through the simulation, this behaviour should be clearly visible, although initially it is expected that congestion will be experienced while in the "slow start" phase. In the basic simulation, the Bandwidth of the Link is 64Kbps, and it's Propagation Delay is 20ms. The Queue has space for 4 packets, where the maximum size of a packet is limited to 512 bytes (the maximum segment size). Therefore, the Queue contains 2048 bytes of space. The space in the Link ("the pipe") is equal to the delay bandwidth product, or 64Kbps * 20ms = 1280 bytes. The maximum amount of data that can be in the network at any one point in time is 3328 bytes, or 7 packets.

Congestion should occur as the window increases beyond this point, represented in Figure X.


Queue Length (Number)
Maximum Queue Length
Drops occur, then queue depletes because of transmitter waiting to detected loss
The nature is cyclic
Time (seconds)
Transmitter has picked up loss and starts (re)transmitting

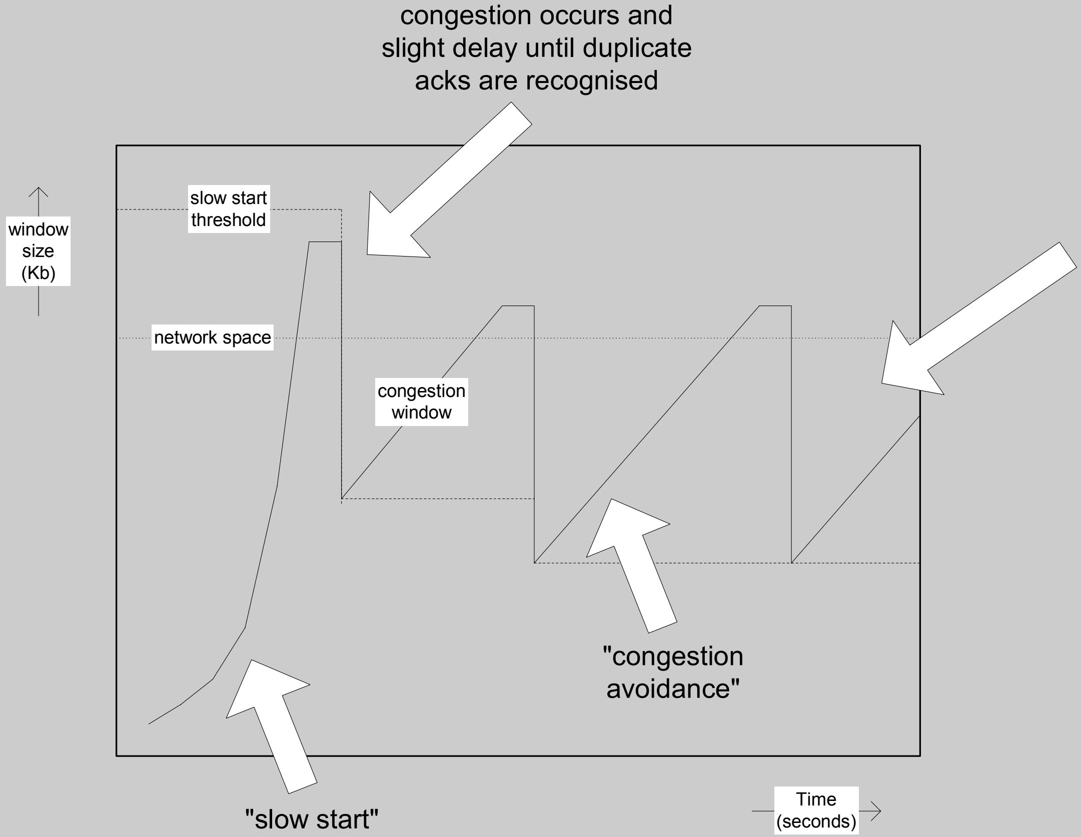
At some point in time, whether in “slow start” or “congestion avoidance”, congestion will occur, and a packet will be dropped. This should be observable on the diagram that displays the length of the Queue, until this point the Queue will have been slowly growing, and the Round Trip Time (RTT) would have also been increasing, as packets are waiting longer in the Queue. Finally, when the Queue has grown to its maximum length, it drops an incoming packet, because it cannot fit. The expected Queue and RTT relationships are shown in Figure X.

<show figure here that has the queue growing, and also the RTT growing>

The transmitter in either one of two ways detects the loss. The classic way is for the TCP retransmission timer to expire, which indicates that some acknowledgements have not been received for the previous data sent. The retransmission timer is tailored to be just more than the experienced Round Trip Time (RTT) as all acknowledgements should be received within an RTT.

BSD 4.4 / Net3 (Berkeley Software Distribution, 1994) as used in the BONeS model, has a mechanism referred to as “fast retransmit” (Stevens, 1994). The operation of TCP is such that an acknowledgement is sent for every received segment (or thereabouts, as TCP has a delayed acknowledgement strategy as well). If a segment arrives out of order, potentially due to a predecessor having been dropped in the network, the returned acknowledgement will be equivalent to the last returned acknowledgement. TCP does not implement selective acknowledgements, so an acknowledgement always indicates the next expected sequence number. Until the “lost” packet is received, the acknowledgement will always ask for it. The receiver will therefore received a number of duplicate acknowledgements, and the “fast retransmit” strategy detects the reception of three duplicate acknowledgements and makes a fair assumption that a packet has been dropped in the network, and therefore that retransmission must occur. However, this assumes that the window size is currently greater than 3 segments; otherwise not enough duplicate acknowledgements can be generated.

In this basic simulation, with a window size greater than 4 segments, loss -- and therefore congestion --, should be detected through “fast retransmit”. On the results obtained, this should be indicated by three acknowledgements of equivalent value, with a retransmission event occurring upon the third, such as illustrated in the Figure X.

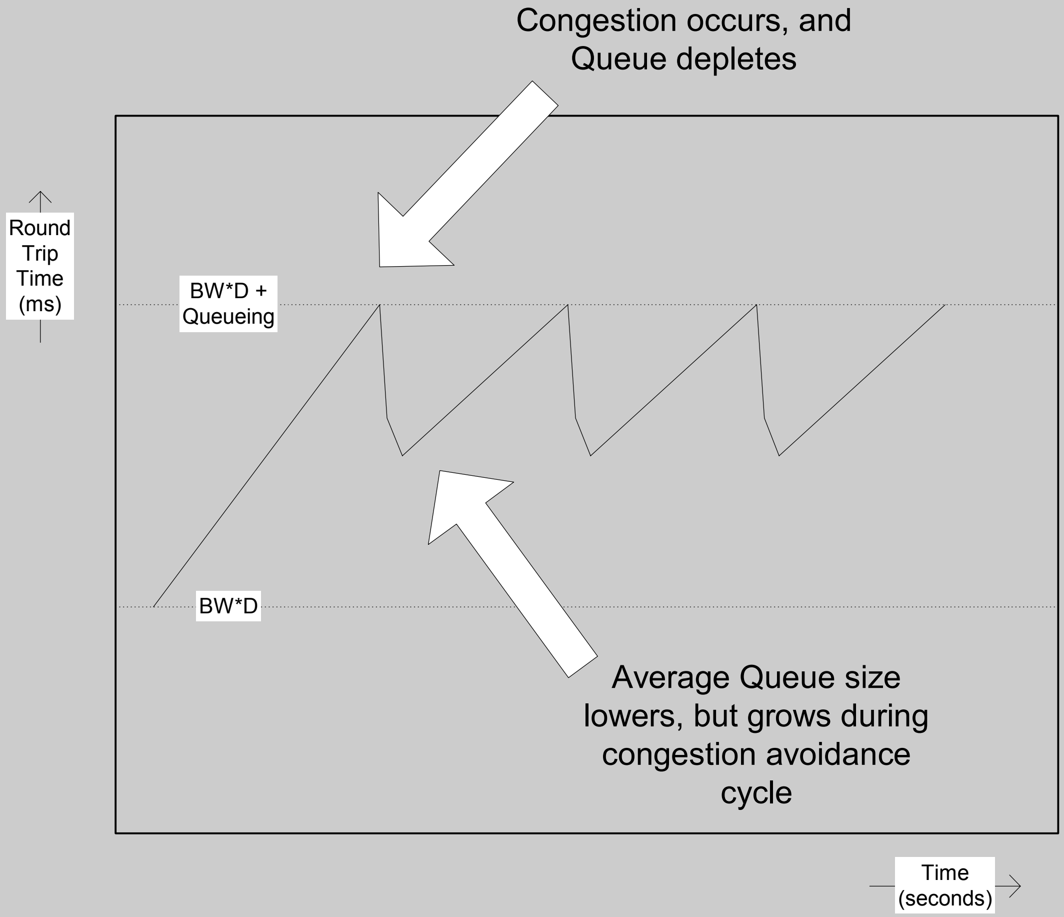


Because loss is an indicator of congestion, a retransmission involves congestion control activity. The behaviour of congestion control is different depending on whether detection has been through the classic or “fast retransmit” mechanism. Under the classic mechanism, the slow start threshold is reduced to half the current congestion window, and the congestion window is reduced to a value of one segment. This behaviour is due to the assumption that a safe operating point is at least half the point at which congestion occurred (Jacobson, 1988), and that with unknown network conditions, the slow start phase should be used.

If loss is detected through “fast retransmit”, then a mechanism called “fast recovery” is employed. The transmitter assumes that although (at least) one packet has been lost by the network, it should not should not slow down too much, but attempt to “keep the pipe full” (Jacobson, 1990). In this case, it first reduces the slow start threshold to half the current congestion window, as in the classic case, but then sets the congestion window to a value of one segment and initiates retransmission. This should then result in the transmission of a single segment to make up for the one that has been lost. After this, the congestion window is set to slow start threshold plus three segment sizes. This reduces the congestion window, but allows it to continue placing data into the network. Upon the reception of subsequent duplicate acknowledgements (presuming that the retransmitted segment has not yet reached the receiver), the congestion window is increased by one segment. Finally, when a non-duplicate acknowledgement is received for the outstanding segment (or, more correctly, for the outstanding segment and all subsequent segments sent in between), the congestion window is set to the slow start threshold, and commences the "congestion avoidance" phase.

In the basic simulation, this situation should occur. The graphs should illustrate the transmission of an "old" segment upon the retransmission event and the change in the slow start threshold and congestion window. Eventually, there is reception of an updated acknowledgement, and the congestion window alters; after which linear increase occurs.

It may now be apparent that there is a periodic nature here, with the congestion window advancing, reaching a maximum, and proceeding through retransmission and then through the process again. This is a defining characteristic of the window based congestion control algorithm that TCP employs (doesn't it look very much like the charge, leakage, and discharge of current in a capacitor?). For the basic simulation, the entire lifecycle of congestion window and slow start threshold should follow this periodic nature, and the relevant graphs would appear as Figure X.



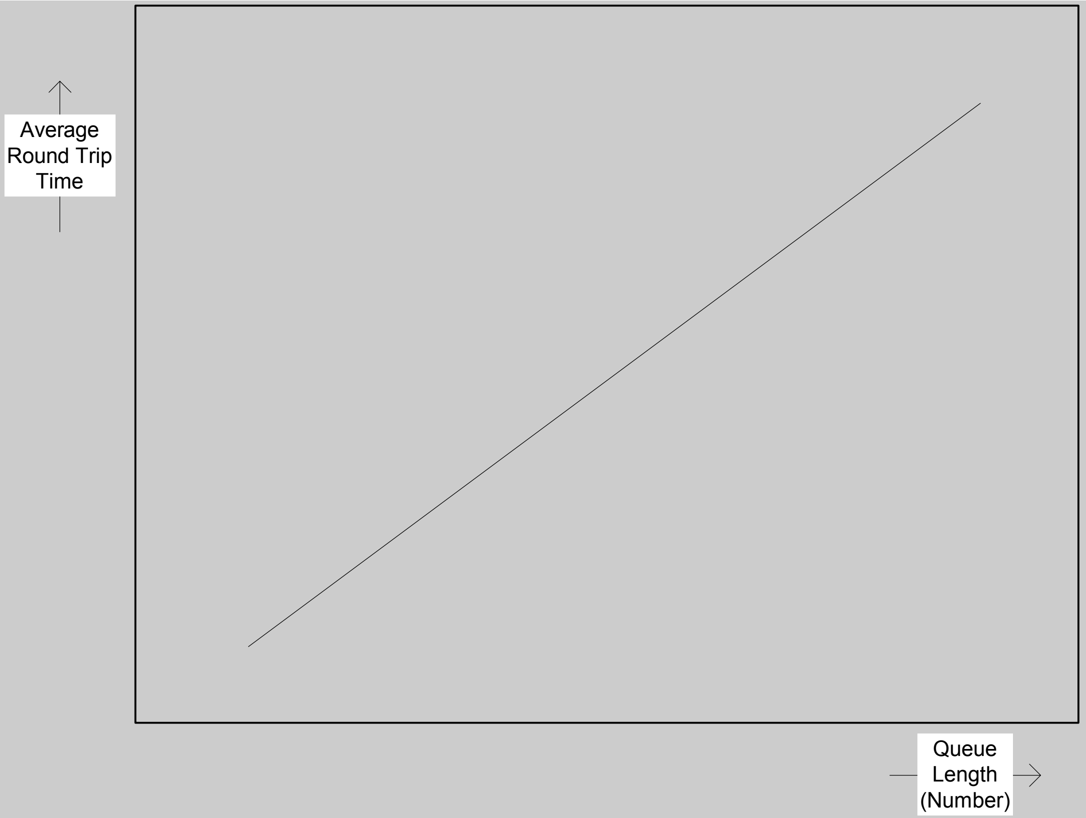
With known network parameters, it is possible to estimate the particular values of these characteristics. If, as previously indicated, the network has a total capacity of 3328 bytes, or 7 segments. Consider that initially, the congestion window is set to one segment. With no queuing delays, it takes 84ms delay through the link (20ms propagation delay, and 64ms transmission delay for 512 bytes), and the acknowledgement takes a conservative 28ms on the way back (20ms propagation delay, and 64ms transmission delay for 64 bytes). These figures must be taken as approximate, because lower layer headers will increase packet sizes; in a practical network, we could only ever make approximations anyway. We can consider a detailed analysis of the time that segments enter the queue, the arrival of acknowledgements and the generation of new segments into the queue, but as an approximation we can consider that the during the first RTT, there is one segment, during the second RTT, there are two segments, during the third RTT, there are four segments, and during the fourth RTT there are eight segments -- in the network. Each RTT will grow by approximately 64ms due to queuing delays[3]. Therefore, it should be after 784ms, or around about the 1-second mark, that congestion occurs.

To predict the periodicity of the congestion window cycle, consider that when congestion occurs and a packet is lost, the retransmission occurs after three duplicate acknowledgements. It returns the window to four segments, and enters congestion avoidance. With four segments in the network, two are in the pipe, and two are in the queue. With each round trip time, the queue increases by one (approximately), so it takes five round trip times until congestion occurs. The round trip time is initially 100ms + 2 * 64ms = 228ms, with two segments in the queue, after which it increases by 64ms each round trip time.

It takes five round trip times until the window reaches 8 and congestion occurs. With four packets in the network, two will be in the pipe, and two will be in the queue, so the round trip time is approximately 100ms + 2 * 64ms = 228ms, after an increase by one, it is 292ms. Therefore, four round trip times are completed after about 1.7 seconds. Because of retransmission delays, and so forth, we can expect the period to lie within the 2 to 3 second mark.

By examination of the results, this information should be visible, and it should be possible to correlate the actual and measured round trip times, as they oscillate between a minimum of 100ms, and a maximum of 420ms. 100ms represents the time for two propagation delays, and delays for transmission of 512 byte and 64 byte segments. 420ms represents this base round trip time, plus additional queuing delay of 5 maximum segments (note that a segment will enter the queue, and wait for the four in front of it to be sent, along with the one that is currently being sent). The round trip time graph can then be expected to look like that shown in figure x.

[3] This is also an approximation, because queuing delays will not occur until we have first exhausted the “space in the pipe”, which is at least one full sized segment.



The queue length will show periodicity correlated with the window, as for each period it will initially contain two segments, and then slowly increase by one until it must reject an incoming segment because it is full. It should look something like that shown in figure x

<the queue diagram>

With the known window behaviour, approximate throughput and retransmission values can be determined. The case is that within each period, only one segment will be lost. This gives rise to an approximate loss of anywhere up to 512 bytes every 2.5 seconds. However, during the same period, the link will be operating at maximum utilisation (as discussed, when a retransmission occurs, the queue still has segments in it, and the addition of new segments and additional duplicate acknowledgements mean that the queue will never deplete). This means that during 2.5 seconds, the link transmits a total of 20kb (at 64kbps), the fact that 512 bytes of this are retransmissions is insignificant. The graphs for data transmission should show a linear increase at approximately 64kbps for the data transmitted (remember, there are lower layer headers), and a 0.5k step every 2.5 seconds for data retransmitted.

<the data diagram>

As mentioned, the link should be fully utilised after transient start up, so we should observe a flat response at 100%.

When the Queue length is varied, we can expect more interesting results. A greater queue length provides more space for segments in the network, so a larger congestion window is possible. There is an immediate problem with this, in that during the initial transient response, the congestion window is doubled for each round trip time, until congestion occurs. With a larger window, the potential situation is that more segments are transmitted into the network and immediately dropped. This actually parallels the case of overshoot in the transient response of electrical circuits, and we could potentially draw an analogy between queuing space and capacitance.

We should therefore tend to see retransmission levels that are, in the initial transient, worse as queue length increase. However, during periodic operation, with linear increase, retransmission levels should be slightly higher, but not significantly. Overall, the relationship between queue lengths and retransmission ratio is expected to be similar to that shown in figure x.

<relationship between queue length and retransmit ratio>

The same throughput is achieved with increasing queue lengths, so we should only tend to see a slight degradation to account for the increase in retransmission levels. The relationship between queue lengths and throughput is shown in figure x.

<relationship between queue length and throughput>

The big loss in increasing queue lengths is the delay introduced into the network, which manifests itself through increased round trip times. It is important to consider here if we did not have “fast retransmit”, then retransmission would only be detected through the retransmission timer, which waits for at least a round trip time. Therefore, under classic TCP, the throughput figures would be even less, due to the wait incurred. The relationship between queue lengths and average round trip times is shown in figure x, the rate of increase can be expected at around the 64ms mark -- to account for the additional extra delay in the queue.

<relationship between queue lengths and avg round trip times>

It can be further noted that without “fast recovery”, the TCP would go back and transmit all previous segments in the window, resulting in a further increase in retransmission levels, and decrease in overall throughput.

3.1.6. Execution of Simulation

The simulation was not executed due to the problems surrounding the unavailability of the BONeS software.

3.1.7. Analysis of Results

No results were gathered from the simulation due to the problems surrounding the unavailability of the BONeS software.

3.1.8. Conclusions (NOT FINISHED)

in this particular scenario, our objectives were concerned with explanation and verification and validation. with results, we could conclude that we had indeed produced a representative model. we can see that the tcp congestion control does work, although it is not perfect due to losses, and the oscillitory nature. this is recognised [ref], and the primary motiviation behind the examination of other mechanisms [refs...]. in more complex networks, we expect to see this behaviour drift, but the core characteristics will remain the same. we do however have a good understanding of the basics.

we can make some conclusions about the effects of increased queue lengths. an often illhad beleif is that increased queueing in a network can be beneficial, but as our expectations tell us, increased queue does nothing more than <x,y,z>

3.2. Multiple TCP Conversations through bottleneck WAN Router

3.2.1. Problem and Objectives

In practice, congestion control measures operate in shared environments where they must interact and co-operate with each other. The first scenario considered the case of a simple TCP conversation, in a point-to-point situation. The second scenario seeks to build upon that by introducing competing parties, and by expanding the topology to a more realistic level. The conversations now will not only interact with network constraints, but also with their peers.

The basic objective in this scenario is to examine the competitive nature of the TCP congestion control mechanisms and their behavioural relationships with other traffic in the network. Originally, this scenario was to involve iteration with various congestion control strategies in an attempt to compare the relative advantages and disadvantages of each one, but these strategies have not been implementation and time limitations have prevented this from occurring.

The problems recognised with co-operating congestion control mechanisms, in general, seem to involve Round Trip Times (RTT), but there are other problems. It is desired to view, examine and explain these problems as they manifest themselves in the scenario executed here. Therefore, *the first objective is to examine the nature of TCP congestion control in a shared and competitive environment.*

In the same manner as the first scenario, this scenario is also intended to serve as a platform to explain the basic behavioural aspects of the TCP congestion control mechanisms. In accordance with the first objective, the focus here is upon those aspects as they relate to co-operation and interaction between conversations. Therefore, *the second objective is to further the explanation of the nature of the TCP congestion control mechanisms.*

It should be noted that the understanding gained through these first two scenarios is particularly important for the last three scenarios, which implicitly assume prior understand of the issues raised and discussed here.

3.2.2. Discussion and Related Work (NOT FINISHED)

competing tcp conversations have been the focus of many studies [ref]. these studies consider the role both the tcp [ref] and the intemrediate systmes [ref] play in the behaviour.

the problems associated with rtt have been looked at by [ref], whereas [ref] has considered how the particulars of the mechanism can cause what is referred to as “phase effects”, where particular conversations in competition can be subject to significant discrimination. interesting enough, [ref] comments that this may be entirely due to the particulars of the simulation environment, and rarely [if ever] present in actual environments.

[ref] .. [ref] has looked at the the effects of intermediate queue policies in depth, and developed “random early detection”.

we have a basic understanding about what should occur, but by the knowledge about our own environment, we can be more detailed in our examination.

3.2.3. Approach

The approach consists of identifying the model, simulation and observations that are required to obtain the objectives.

Model

The model consists of a typical WAN environment having a three LANs interconnected through a central WAN Router. Each LAN has a number of Hosts, which are internally connected to a LAN Router. The Links between the LAN Router and the WAN Router provide the primary resource constraint in the network. The queues in the WAN Router provide the secondary resource constraint. This provides the general case of high speed LANs, interconnected by a lower speed WAN.

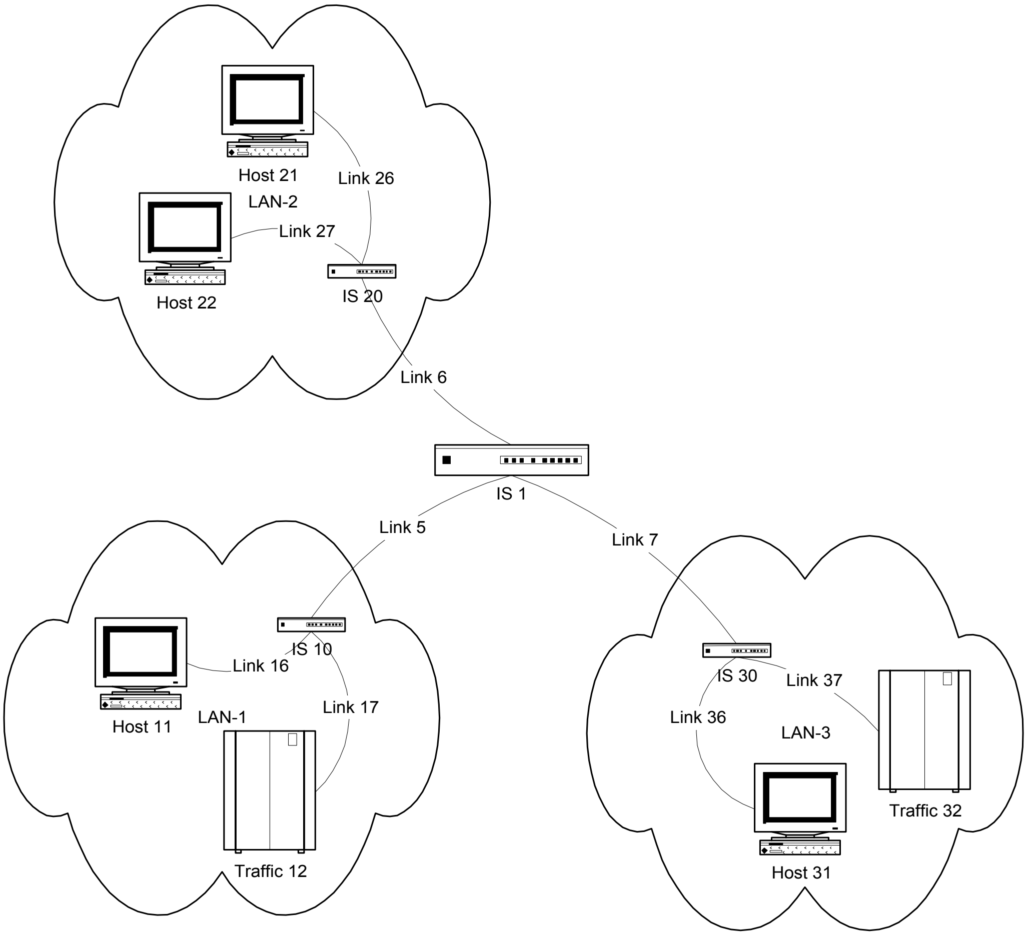


Figure 2-3.4. Simulation Model: Multiple TCP Conversation WAN

A number of parameters in the model are fixed. The LANs employ Links with Bandwidth and Propagation Delay of 10Mbps and 1ms, in order to represent a typical Ethernet environment. The LAN Routers are configured with appropriate routing entries and their queue length and discipline is not of concern, so is set to 15 (packets) and Drop Tail, respectively. Observation is made to ensure that the LAN Routers do not become congested. The WAN Links have a Bandwidth of 64Kbps, and

Propagation Delay of 20ms, to represent typical low speed WAN connections. The WAN Router is set to use a queue length of 8 and a discipline of Drop Tail. The routing table for the WAN Router is set to ensure that all Hosts and Traffic used in the simulation can communicate with each other. Iterations of the simulation alter the WAN Links and WAN Router parameters.

Basic Simulation

The basic simulation starts with a conversation between Host 11 in LAN 1 and Host 21 in LAN 2. This conversation passes through Link 5 and Link 6, and carries one-way traffic (a transfer of a large unit of data) between Host 11 and Host 21. The return traffic consists only of acknowledgements. This conversation runs for a time sufficient to allow it to reach steady state, for which 30 seconds should be appropriate.

During this phase, observations are made of the TCP transmitter in Host 11, and the queue for Link 6 in the WAN Router. These observations should be similar to those seen in the first scenario and are not important in this scenario.

After 30 seconds, a second conversation is started between Host 31 in LAN 3 and Host 22 in LAN 2. This conversation passes through Link 6 and Link 7, and carries one-way traffic (a transfer of a large unit of data) between Host 31 and Host 22. The return traffic consists only of acknowledgements. This conversation runs for 90 seconds.

During this phase, important observations are made relating to the co-operation between both TCP transmitters, and therefore the TCP transmitters in Host 11 and in Host 31 are examined. The queue for Link 6 in the WAN Router now provides detail on occupancy as a whole, and for each conversation. To ensure the correct operation of the simulation, the queues for the WAN Links at the LAN Routers are monitored along with the utilisation of the WAN Links themselves.

The observations will tend to show congestion occurring at the WAN Router, and then each conversation attempting to reach and maintain a stable (but possibly oscillating) operating points.

At 120 seconds, when sufficient observation has been made of phase 2, Traffic is generated between Host 32 in LAN 3 and Host 12 in LAN 1. The traffic (uniformly random data at poisson time intervals) will traverse Link 5 and Link 7, in both directions. This is allowed to run for 30 seconds.

During this phase, the same observations are made as in the previous phase, although more care is taken to ensure that the queues for the WAN Links in LAN Routers don't become congested. Additional observations are made of the queues in the WAN Routers for Links 5 and Links 7, but only for data on the two main TCP conversations.

The simulation is stopped after 150 seconds.

Variations

There is interest in examining the effects of RTT values, particularly with respect to bias effects. To achieve this, the basic simulation is run by iterating the Propagation Delay in Link 7. This causes the conversation between Host 31 in LAN 3 and Host 21 in LAN 2 to incur an RTT different to that of the other conversation. The iteration can use values from 10ms to 200ms in increments of 10ms.

To examine the involvement of the Router in terms of its Queue Discipline, iterations are performed using either Drop Tail or Random Drop. These are carried out in addition to the RTT iteration, i.e. providing an additional indication of how particular RTT effects manifest themselves depending on the particular discipline in use.

Finally, as an attempt to better judge the effects of the introduced background traffic, iterations are performed to alter its characteristic. To enhance acknowledgement compression, it is desired to have potentially larger delays in the queues, however it is not desired to have congestion occur; therefore the iteration occurs on the size of the packets generated as traffic, not upon the number (i.e. frequency) generated.

Observations

As mentioned, the primary observations are performed on Host 11, Host 31 and in WAN Router. Secondary observations are made of WAN Link utilisation and the lengths of the Queues in the LAN Routers for the WAN Links.

For Host 11 and Host 31, TCP characteristics as gathered in the first simulation are observed. For the WAN Router, the lengths of the queues are observed, both in an absolute sense and as for each conversation.

3.2.4. BONeS Simulation Design

Transfer from an abstract approach into a simulation first requires the construction a BONeS simulation module. Probes are then placed into this module to capture data during the simulation, noting that for all runs the same probe configuration is used (this is done for simplicity). The operation of the Basic Simulation, with details about Parameters and execution script, is given, after which the modifications are described for each subsequent iteration.

Every simulation is run with iteration of the “Global Seed” Parameter, at least three times. This particular aspect is not explicitly outlined because it is carried out so that visual observation can be made to ensure that results are correct. It is fortunate that the automated capability of BONeS allows for this to be carried out quickly and effortlessly.

3.2.4.1. Topology

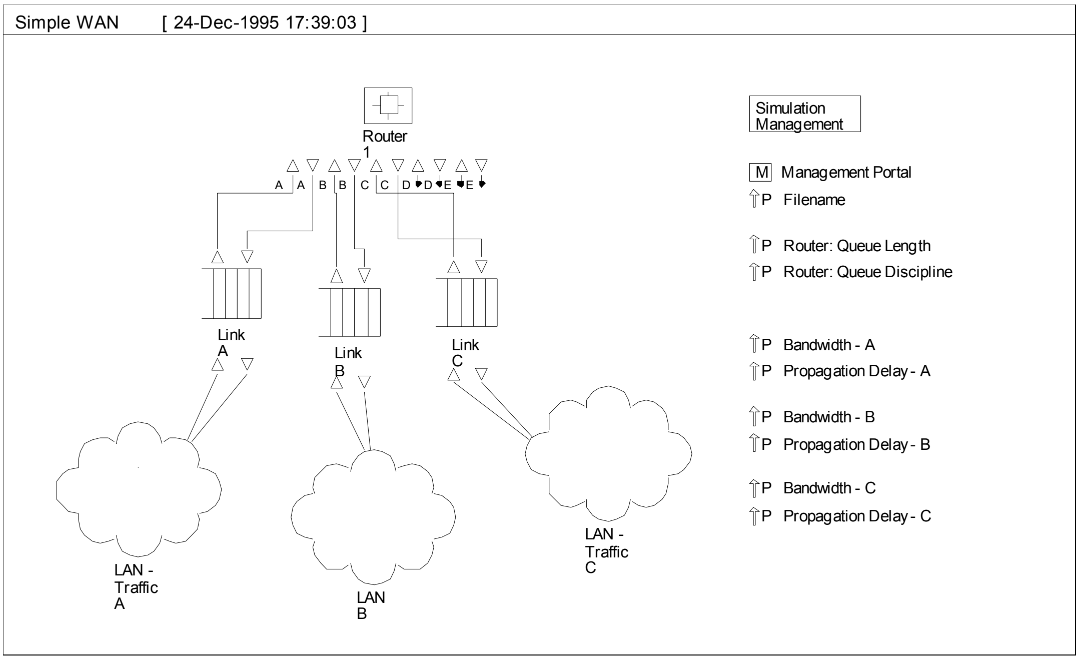


Figure 2-3.5. Simulation Topology: Multiple TCP Conversation WAN

The approach is translated into an actual BONeS simulation first through the construction of a simulation Module using the components developed on Part 1 of this thesis. The parameters relevant to the simulation are visible in the figure.

3.2.4.2. Post Processing and Probe Placement

To construct information used in the analysis, Probes can be placed into the simulation using the BONeS Simulation Manager; once placed, they are then used in the Post Processor to generate graphs. The approach taken here is to first identify the particular graphs that indicate critical information for analysis, and then to determine which Probes must be placed, and where they must be placed.

3.2.4.2.1. Basic Simulation

For the basic simulation, the graphs illustrate the lifecycle activity in the host and the network. Interest is with both TCP transmitters, and the queuing information from the WAN Router.

TCP Window Information	
For	Host 11, Host 31
Purpose	To show the detailed attributes of the TCP congestion control algorithm, as it alters during the course of the simulation. In addition, events that are correlated with TCP congestion control activity are also captured.
X Axis	(Seconds): Time
Y Axis	(Bytes): Congestion Window, Slow Start Threshold, Unacknowledged Data (No Units): Retransmission Events, Timer Expiries
Probes	TCP Probes are used, and they are placed into Host 11 and Host 31's Transport Layer.

TCP Computed and Actual Round Trip Time (RTT) Information	
For	Host 11, Host 31
Purpose	The RTT plays an important role in TCP congestion control. However, as it is estimated, observations of the actual RTT should also be made.
X Axis	(Seconds): Time
Y Axis	(Milliseconds): RTT Value, RTT Value +RTT Variance, RTT Value - RTT Variance, Actual RTT
Probes	TCP Probes are placed into Host 11 and Host 31’s Transport Layer. The Actual RTT is obtained by placing a probe into Host 11 and Host 31’s Transport Layer to extract the timing information from a received acknowledgement.

WAN Router Queue Information	
For	WAN Router for Host 11, Host 31, Host 21, Host 22
Purpose	The queue drops packets, and affects the RTT for packets. Its behaviour can be correlated with that of TCP congestion control.
X Axis	(Seconds): Time
Y Axis	(Integer Value): Queue Length, Queue Drops
Probes	Queue Probes are placed into the WAN Router’s Network Layer to capture the total queue length, and the queue usage for the particular destination addresses given. The Network Layers are those that connect to the link that is directed towards the destination in question.

Transport Layer Data Transmission	
For	Host 11, Host 31
Purpose	The qualitative information about a conversation is related to its throughput and retransmission levels. The number of transmitted and retransmitted bytes is also affected, and can be correlated with, TCP congestion control activity.
X Axis	(Seconds): Time
Y Axis	(Kilobytes): KB Transmitted, KB Retransmitted
Probes	TCP Probes are placed into Host 11 and Host 31’s Transport Layer.

Transport Layer Data Transmission (95% confidence level)	
For	Host 11, Host 31
Purpose	For greater confidence in the simulation results, a confidence plot using different initial random seeds is used. The information best used on a confidence plot is the throughput and retransmit levels, as the assumption is that they are relevant equivalent for a given scenario. Window and Queue information is more highly variant, and subject to phase differences.
X Axis	(Seconds): Time
Y Axis	(Kilobytes): KB Transmitted, KB Retransmitted
Probes	TCP Probes are placed into Host 11 and Host 31’s Transport Layer.

WAN Link Utilisation	
For	Link 5, Link 6, Link 7
Purpose	Because of retransmission timeouts and other events, the link may not always be fully utilised, where under ideal conditions it should always be.
X Axis	(Seconds): Time
Y Axis	(Percentage): Utilisation
Probes	Probes are placed into Link 5, Link 6 and Link 7. They capture the sum of all packet lengths passed through the link over the total capacity made available by that link according to the length of time in the simulation.

LAN Router Queue Information	
For	LAN 1 Router, LAN 2 Router, LAN 3 Router
Purpose	These routers should not play a significant role in the simulation, so they are observed to ensure that they don't.
X Axis	(Seconds): Time
Y Axis	(Integer Value): Queue Length, Queue Drops
Probes	Queue Probes are placed into the LAN 1 Router, LAN 2 Router and LAN 3 Router's Network Layer to capture the total queue length, and the queue usage for the particular destination addresses given. The Network Layers are those that connect to the link that is directed towards the destination in question.

3.2.4.2.2. RTT Iteration

When the RTT is iterated for one of the conversations, various relationships are measures to assess the impact.

Average Queue Share versus. RTT Ratio	
For	WAN Router, Host 11 and Host 31 into Link 6
Purpose	When the two conversations are competing, they share the queue. This share may alter depending upon RTTs
X Axis	(Real Value): RTT Ratio
Y Axis	(Integer Value): Average Queue Length
Probes	The Probes used are those from the Basic Simulation. The RTT ratio is taken by computing the average RTT for both conversations, and dividing them. The Queue Length is the average length computed for each particular conversation across the life of the simulation.

Throughput versus. RTT Ratio	
For	Host 11 and Host 31
Purpose	Throughput is related to RTT ratio.
X Axis	(Real Value): RTT Ratio
Y Axis	(Integer Value): Throughput
Probes	The Probes used are those from the Basic Simulation. The RTT ratio is taken by computing the average RTT for both conversations, and dividing them. The throughput is taken as the total number of bytes transmitted for the conversation over the time of the conversation, for each simulation run.

Retransmission Ratio versus. RTT Ratio	
For	Host 11 and Host 31
Purpose	Throughput is related to RTT ratio.
X Axis	(Real Value): RTT Ratio
Y Axis	(Integer Value): Retransmission Ratio
Probes	The Probes used are those from the Basic Simulation. The RTT ratio is taken by computing the average RTT for both conversations, and dividing them. The retransmission ratio is constructed for each conversation in each simulation run.

3.2.4.2.3. Traffic Level Iteration

When the traffic level is iterated, the effects on performance are examined.

Throughput versus. Traffic Level	
For	Host 11 and Host 31
Purpose	As the level of background traffic increases, the throughput levels may suffer due to the share with acknowledgements.
X Axis	(Kilobytes per second): Traffic Level
Y Axis	(Integer Value): Throughput
Probes	The Probes used are those from the Basic Simulation. The RTT ratio is taken by computing the average RTT for both conversations, and dividing them. The throughput is taken as the total number of bytes transmitted for the conversation over the time of the conversation, for each simulation run.

Retransmission Ratio versus. Traffic Level	
For	Host 11 and Host 31
Purpose	As the background traffic increases, more retransmissions may occur, due to the loss of acknowledgements or for other reasons.
X Axis	(Kilobytes per second): Traffic Level
Y Axis	(Integer Value): Retransmission Ratio
Probes	The Probes used are those from the Basic Simulation. The RTT ratio is taken by computing the average RTT for both conversations, and dividing them. The retransmission ratio is constructed for each conversation in each simulation run.

Average RTT versus Traffic Level	
For	Host 11 and Host 31
Purpose	As the background traffic increases, the average RTT should increase due to the increased loading for returned acknowledgments
X Axis	(Kilobytes per second): Traffic Level
Y Axis	(Milliseconds): Average RTT Value, Average RTT Variance, Average actual RTT
Probes	The Probes used are those from the Basic Simulation. The graph is constructed by taking the average RTT for the conversation for each simulation run.

3.2.4.3. Execution: Basic Simulation

In the basic simulation, the Parameters must be configured using the BONeS Set Parameters Dialog. One such parameter is the Management Script. There is no iteration in the basic simulation.

3.2.4.3.1. Parameters

The parameters correspond to the values discussed in the Approach.

Parameter	*Value*	*Description*
Filename	multiple.txt	The file contains the Management Script
WAN Router: Queue Length	4	A relatively typical length.
WAN Router: Queue Discipline	Drop Random	In the Basic Simulation, choose the best.
Link 1: Bandwidth	64kbps	Model an ISDN B Channel.
Link 1: Propagation Delay	20ms	Model a typical delay.
Link 2: Bandwidth	64kbps	Model an ISDN B Channel.
Link 2: Propagation Delay	20ms	Model a typical delay.
Link 3: Bandwidth	64kbps	Model an ISDN B Channel.
Link 3: Propagation Delay	20ms	Model a typical delay.

3.2.4.3.2. Management Script

The Management Script is broken up into a number of steps according to the outline given in the Approach. The place at which alterations are made for the Traffic Level iteration is highlighted in bold.

Step 1: Initial configuration at Time 0

Set Routing Entries for the Router at the WAN -- The routing entries need to indicate that the particular Hosts within each LAN are reachable via their respective Links.

```
0 ⇒ Router 1 : Set Route Entry (Addr: Host 11, If: Link 1, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Traf 14, If: Link 1, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 21, If: Link 2, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 22, If: Link 2, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Traf 24, If: Link 2, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 31, If: Link 3, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Traf 34, If: Link 3, Cost: 1)
```

Set Routing Entries for the Router in LAN 1 -- The routing entries need only be set for the Hosts that communicate to and from LAN 1.

```
0 ⇒ Router 10 : Set Route Entry (Addr: Host 11, If: Link 16, Cost: 1)
0 ⇒ Router 10 : Set Route Entry (Addr: Traf 14, If: Link 19, Cost: 1)
0 ⇒ Router 10 : Set Route Entry (Addr: Host 21, If: Link 1, Cost: 1)
0 ⇒ Router 10 : Set Route Entry (Addr: Traf 34, If: Link 1, Cost: 1)
```

Set Routing Entries for the Router in LAN 2 -- The routing entries need only be set for Hosts that communicate to and from LAN 2.

```
0 ⇒ Router 20 : Set Route Entry (Addr: Host 21, If: Link 26, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 22, If: Link 27, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Traf 24, If: Link 29, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 11, If: Link 2, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 31, If: Link 2, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Traf 34, If: Link 2, Cost: 1)
```

Set Routing Entries for the Router in LAN 3 -- The routing entries need only be set for Hosts that communicate to and from LAN 3.

```
0 ⇒ Router 30 : Set Route Entry (Addr: Host 31, If: Link 36, Cost: 1)
0 ⇒ Router 30 : Set Route Entry (Addr: Traf 34, If: Link 39, Cost: 1)
0 ⇒ Router 30 : Set Route Entry (Addr: Host 22, If: Link 3, Cost: 1)
0 ⇒ Router 30 : Set Route Entry (Addr: Traf 14, If: Link 3, Cost: 1)
```

Step 2: Establishment of TCP conversation between Host 11 in LAN 1 and Host 21 in LAN 2 at Time 0

Set Initial Sequence Numbers for Host 11 and Host 21.

```
0 ⇒ Host 11 : Set Parameters (ISN: 12345678)
0 ⇒ Host 21 : Set Parameters (ISN: 12345678)
```

Request Host 11 to Connect Session to Host 21, and Host 21 to Connect Session to Host 11.

```
0 ⇒ Host 11 : Connect Session (Addr: Host 21)
0 ⇒ Host 21 : Connect Session (Addr: Host 11)
```

Instruct the Generator on Host 11 to produce a single Constant unit of data.

```
0 ⇒ Host 11 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value:
10000000))
```

Step 3: Establishment of TCP conversation between Host 31 in LAN 3 and Host 22 in LAN 2 at Time 30

Set Initial Sequence Numbers for Host 31 and Host 22.

```
30 ⇒ Host 31 : Set Parameters (ISN: 12345678)
30 ⇒ Host 22 : Set Parameters (ISN: 12345678)
```

Request Host 31 to Connect Session to Host 22, and Host 22 to Connect Session to Host 31.

```
30 ⇒ Host 31 : Connect Session (Addr: Host 22)
30 ⇒ Host 22 : Connect Session (Addr: Host 31)
```

Instruct the Generator on Host 31 to produce a single Constant unit of data.

```
30 ⇒ Host 31 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value:
10000000))
```

Step 4: Establishment of Traffic between Traffic 14 in LAN 1 and Traffic 34 in LAN 3 at Time 90

Set Address Lists on Traffic 14 for Traffic 34, and on Traffic 34 for Traffic 14

```
90 ⇒ Host 14 : Set Address List (Num: 1, Addr: 34)
90 ⇒ Host 34 : Set Address List (Num: 1, Addr: 14)
```

Instruct the Generator on Traffic 14 and Traffic 34 to produce poisson units of data -- the value used here is subject to iteration, i.e. the Length.

```
90 ⇒ Host 14 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: POISSON, Lambda: X), Space (Type: CONSTANT, Value:
<ITER>))
90 ⇒ Host 34 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: POISSON, Lambda: X), Space (Type: CONSTANT, Value:
<ITER>))
```

Step 5: Terminate the simulation at Time 150

Stop.

```
150 ⇒ :
```

The Management script is constructed by translating these pseudo operations using the information provided in Part 1. This is not provided here, as it is cryptic and pointless.

3.2.4.4. Execution: RTT Iteration Simulation

The same Parameters and Management script are used as in the Basic Simulation, however for the Propagation Delay for Link 3, a BONeS iteration dialog is selected. This dialog is instructed to step through the Propagation Delay from values 10ms to 200ms. This procedure is carried out twice, first for a Queue Discipline of “DropTail” and second for a Queue Discipline of “DropRandom”.

3.2.4.5. Execution: Traffic Level Iteration Simulation

The same Parameters are used as in the Basic Simulation, however a number of Management Scripts are created, to iterate the “Space” Parameter for the Statistical Generator between 1 and 256.

3.2.5. Expectations (NOT FINISHED)

During the initial phase of the simulation, the TCP conversation between Host 11 and Host 21 will exhibit the same characteristics as described in the expectations of the previous simulation. This includes the transient start up, and periodic steady state response.

The second conversation commences operation at 30 seconds; by this time the first conversation will well and truly be oscillating around the operating point of the network. Although it is possible to consider the network parameters and compute transient and steady state response characteristics, this tends to be problematic as a slight deviance in our figures will result in a potentially entirely different value for the window at the 30 second mark.

The second conversation will immediately inject segments into the network, as it will commence exponential increase of the congestion window. Congestion should occur within one or two round trip times, which for this network can be considered approximately 80ms in propagation delays (two double WAN link propagation delays) plus 128ms in transmission delays (two WAN link propagation delays), plus (say) queueing delay of two segments in the WAN Router, at 128ms: a total of approximately 350ms (ignoring transmission delays due to acknowledgements). These figures are approximate, for the very good reason that the LANs will also introduce their own (albeit small) delays. Therefore, after some 500ms of the second conversation being introduced, both should experience congestion. Our observed results for the TCP transmitters should like similar to those shown in figure x at this point.

<tcp transmitters>

With the first conversation consuming more space than the first, it is more likely that it will incur loss first, however this is not definite but made more likely through the use of Random Drop (Floyd & Jacobson, 1993). We can make some estimations here. Consider that the first conversation would have had full use of the network. With the given network parameters, this corresponds to 320 bytes of space in the WAN pipes (1 segment), and 4096 bytes of space in the Routers (4 segments in each of the LAN Router and WAN Router), giving a total space of about 4500 bytes or 8 segments. If this is the maximum space available, then the first conversation's slow start threshold should be approximately 5 segments, and its congestion window will be anywhere between 5 and 8 segments, say 7 segments.

The second conversation will incur congestion with a lower congestion window, say 2 segments. Both conversations will then half their slow start threshold, to say 3 segments and 1 segments respectively. Although our observations will not have these exact figures, the approximate magnitudes should be apparent, the exact figures are not are important as the basic concept. Both transmitters will then proceed through the linear increase of congestion avoidance. Consider that they will have a roughly equivalent round trip time, so their congestion windows will increase at the same rate. So, after the first round trip time, their windows will be 4 and 2 respectively, then 5 and 3, then 6 and 4 and possibly 7 and 5 before congestion occurs. When they are halfed, they retreat to 3 and 2 respectively. It is clear than after each epoch, they tend to become more fairer in their use of the bottleneck. The observations of the congestion window and slow start theshold during this equilibrium attainment period should look like this shown in figure x.

<converations coming to equailibvrum>

The observations from the router's queue will tend to support this by showing a gradual share. With the small queue size in the basic simulation, this behaviour will not be as apparent as it is with the larger queue sizes. When the queue length is larger, the syncronisation period will be larger (due to the increased round trip time, which results in larger periods as shown in first imulation), and therefore equilibrium will take longer to occur. This indicates another case in which increased queueing impacts upon performance.

The results we obtain for actual and retransmitted data levels will be correlated with the congestion window, in that the first conversation will gradually lose throughput until both it and the second conversation oscillate around the same value. As found through the first simulation, retransmission levels should also be correlated with transmission levels due to the larger congestion windows in operation.

Our next interest is with variations in the propagation delay on Link 7. When this occurs, the second conversation will be subject to greater round trip times than the first conversations. It was shown that with equivalent round trip times, both transmitters increase their windows equallly during congestion avoidance, however with different round trip times, these rates of increase will also differ. Hence, when congestion occurs, the conversation with the larger round trip time will have gained less than the other conversation. It is expected that a fair equilibrium is never obtained.

When executing these iterations, the observations gained should tend to reflect a case that with greater difference in round trip times, the available space given to one conversation will be proportionally different to that given to the other. The conversation with the larger share of the space will, however, suffer a greater level of retransmissions in accordance with its greater share. Therefore, our observations for queuing shares, retransmission ratios and throughputs as a function of the queue length are shown in figure x.

<show diagram of rtt vs q share, rtt vs retx, rtt vs. thru>

The introduction of background traffic to the conversation is specific designed so that it does not affect the forward direction of transfer, therefore it only impacts upon the reverse direction of each conversation. The reverse direction of each conversation does not carry traffic, but carries acknowledgements. With introduced background traffic, the acknowledgements will be subject to additional queueing delay.

Because of this, the average round trip times for each conversation should increase, and the variance in the round trip times should also become larger. The immediate impact upon the transmitters is expected to be a drop in throughput, as the TCP uses received acknowledgements to generate new traffic, and suddenly all acknowledgements are subject to additional delay. However, this is only a small transient set back. The expected relationship betwene round trip times and traffic levels is shown in figure x.

<round trip times vs. traffic levels>

The main behaviour we expect to see is that acknowledgements will not be received with regular spacing. Without any traffic in the reverse direction, all acknowledgements generated by the receiver arrived through the network with the same spacing provided by the receivers. Due to the constraints within the network, the receivers can only receive their data packets (in the forward direction) at regular intervals, therefore the reverse acknowledgements are at such regular intervals. When the spacing is not regular, we expect to see a condition known as ack compression [ref].

With background traffic, an acknowledgement may arrive into the queue and be subject to no delay, significant delay or, in the worst case, it may be dropped due to congestion. If the acknowledmgenet is delayed, then it will obviously be closer to the acknowledgement that follows it. If the acknowledgement is dropped, then the subsequent acknowledgement subsumes the original acknowledgement. The transmitter, upon reception of the acknowledgment, will be able to transmit data straight away.

The closer the acknowlegemnts are toghether, or the larger they are, then the more the transmitter will place into the network at the one point in time. The result will tend to be that instead of generating regularily spaced segments that interleave with those of the other conversation, the transmitter will generate bursts of segments, which have more potential to overflow the queues in the network.

It should be remembered that although the network is capable of supporting a specific amount of data, it supports this data spread out through the network, not at one particular point within the network. The ack compression is expected to disrupt the even distribution of segments within the network and therefore increase levels of retransmission, and in general result in a reduction of throughput. The expected relationship is shown in figure x.

<throughput vs. traffic level>

<retransmissions vs. traffic level>

3.2.6. Execution of Simulation

The simulation was not executed due to the problems surrounding the unavailability of the BONeS software.

3.2.7. Analysis of Results

No results were gathered from the simulation due to the problems surrounding the unavailability of the BONeS software.

3.2.8. Conclusions (NOT FINISHED)

The primary objective was concerned with using this simulation more as tool for validation and verification. As no actual simulation has been carried out, it was not possible to make this assessment, however through the expectations gathered, we can conclude that the simulation will exhibit important TCP congestion control characteristics.

It is expected that when multiple conversations compete, bias to conversations with shorter round trip times will occur, and performance losses will be incurred by those conversations wither larger round trip times. This is consistent with the results from other work [ref].

When background traffic is introduced, the acknowledgement compression is expected to occur, which tend to better reflect actual environment conditions. The result of acknowledgement compression will be increased cases of bursty traffic emanating from the TCP transmitter, and therefore decreased levels of performance due to the resulting increase of congestion in the network.

3.3. Single TCP Conversation in Multiple-Path, Dynamically Routed WAN

3.3.1. Problem and Objectives

This scenario concerns itself with the first major issue identified in relation to the Transmission Control Protocol's (TCP) congestion control mechanisms as they apply in Wide-Area Network (WAN) environments.

The concern is based around the knowledge that the characteristics of WANs are changing from those that existed at the time the TCP and its congestion control measures were devised and instrumented. In particular, the size and complexity of WANs is increasing, leading to situations where an individual conversation may now traverse different paths and be subject to different conditions during its lifetime. In particular, these conditions change within a single Round Trip Time (RTT) of the conversation.

Through the previous scenarios and related work, it has been recognised that RTTs play a significant role in the operation of TCP and its congestion control mechanisms, mostly through the closed loop feedback aspect of TCP congestion control. Problems related to RTTs include fairness bias, and acknowledgement compression. These prove detrimental to the qualitative aspects of a conversation.

It is suspected that a WAN environment with multiple paths, employing some form of dynamic routing (such as selecting a path depending upon localised congestion conditions) will cause packets within a single conversation to traverse different paths, and therefore be subject to RTTs with a high variance. In addition, the incidence of out of order delivery will become more frequent, and returned acknowledgements will not provide the regular clocking that TCP congestion control requires.

Existing work has not adequately addressed environments of this nature, and in general has focused upon relatively simple networks -- this is a well recognised problem. It may be the case that the effects introduced by these complex network scenarios, can be modelled by simpler network scenarios by through traffic effects. This is intuitive, but as yet has not been examined in detail

Therefore, *the objective of is to examine the effects of multiple paths and dynamic routing, and to determine the impact it has on the operation of TCP's congestion control mechanisms.*

3.3.2. Discussion and Related Work (NOT FINISHED)

through investigation, it does not appear that this particular problem has been addressed before. we are aware of what rtt effects can do the tcp congsetion control through the previous simulations. the closest work applicable is that which considers the effects of link failures.

link failures are in some respects similar, but ...

it is acknowledged that the case of more complex network should reciver greater attention.

3.3.3. Approach

The approach consists of identifying the model, simulation and observations that are required to obtain the objectives.

Model

The Model consists of two LANs, separated by a WAN environment with rich connectivity. Within the WAN environment, there are a number of Routers, each of which has an associated Traffic generator used to represent other Traffic in the network, which is not attributed to the two LANs under investigation. The WAN Routers are configured in such a way that multiple paths can be selected between the two LANs.

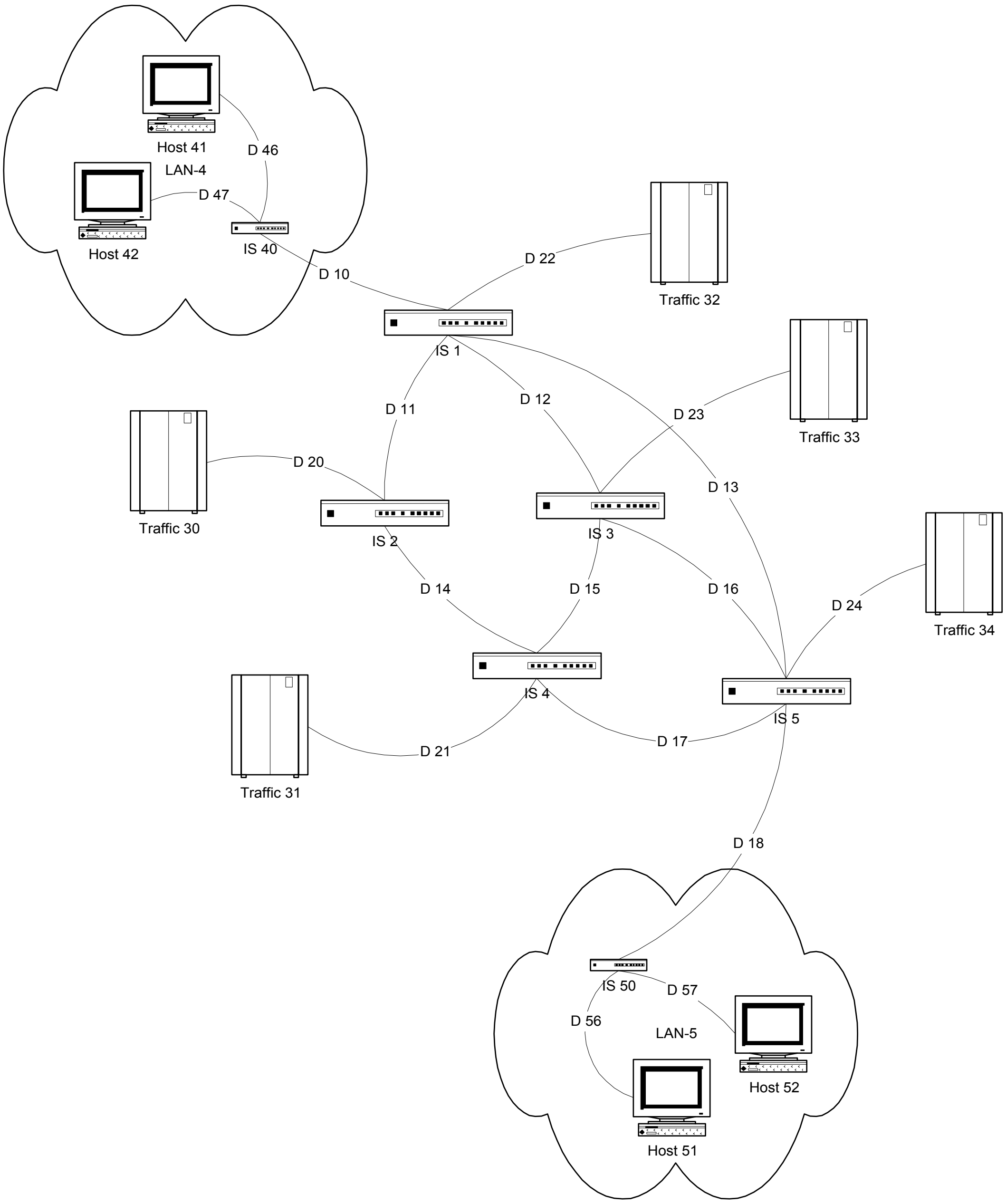


Figure 2-3.6. Simulation Model: Single TCP Conversation WAN

Most parameters are fixed. The LANs are modelled as high speed Ethernet LANs; therefore they have Link Bandwidth and Propagation Delays of 10Mbps and 1ms respectively. The queue length and discipline in the LAN Routers is not important, as they do not have roles in the simulation. Within the network, all WAN Links have Bandwidth and Propagation Delays of 64Kbps and 20 ms respectively: modelling an ISDN environment. The queue lengths and disciplines are set to 8 (packets) and Drop Random respectively, noting that these two parameters will be iterated.

The important aspect of the model is the interconnectivity. A number of paths exist between the two LANs, requiring configuration of the WAN Routers. The WAN Routers are subject to queue loading from the LAN and from the Traffic sources; the Traffic sources play a virtually important role in this respect. The WAN Router selects a path depending on queue loading for the interface associated with that path. The WAN Routers are configured, and as a result the potential paths are as such.

Path	Propagation Delay
Link 10, 13, 18	3x
Link 10, 12, 16, 18	4x
Link 10, 12, 15, 17, 18	5x
Link 10, 12, {15, 15}, 16, 18	(4 + 2n)x
Link 10, 12, {15, 15}, 15, 17, 18	(5 + 2n)x
Link 10, 11, 14, 17, 18	5x
Link 10, 11, 14, 15, 16, 18	6x
Link 10, 11, 14, {15, 15}, 17, 18	(5 + 2n)x
Link 10, 11, 14, {15, 15}, 15, 16, 18	(6 + 2n)x

Basic Simulation

The basic simulation starts with a conversation between Host 11 in LAN 1 and Host 21 in LAN 2. This conversation carries one-way traffic (a transfer of a large unit of data) between Host 11 and Host 21. The return traffic consists only of acknowledgements. This conversation runs for a time sufficient to allow it to reach (a reasonable) steady state, for which 60 seconds should be appropriate.

During this phase, observations are made of the TCP transmitter in Host 11, and in Host 21. This particular scenario requires observation of the TCP receiver's reassembly queue size and -- critically -- the TCP transmitter's RTT values. The queue in every WAN Router is also observed, to determine the path taken by the conversation, and the utilisation of all WAN Links by the conversation is also observed.

At 60 seconds, when sufficient observation has been made of Phase 1, traffic is generated from all Traffic sources. The traffic (uniformly random data at poisson time intervals) will traverse all Links and affect all queues. This is allowed to run for 120 seconds.

During this phase, the same observations are made as in the previous phase, however it is expected that average queue occupancy and total Link utilisation increases. Care must be taken to ensure that the conversation between Host 11 and Host 21 is subject to switching by the Routers due to queue loading effects.

The simulation is stopped after 180 seconds.

Variations

There is interest in altered queue lengths, to examine the impact upon performance due to the increase RTT values experienced through the network. As such, the first variation consists of execution the basic simulation with iterations on the WAN Router queue lengths. These lengths are iterated between 1 and 64.

To examine the effects of the introduced background traffic, iterations are performed to alter its characteristic. This is desirable to increase the level of congestion and delay in the network along with the effect that occurs in relation to dynamic routing. The iteration alters the frequency of traffic generated.

Observations

As mentioned, the primary observations are performed on Host 11 and Host 21 and in the WAN Routers. Observations are also made of WAN Link utilisation.

For Host 11, TCP characteristics as gathered in the first simulation are observed. For Host 21, only the size of the reassembly queue and the network layer's hop count field is of importance, in an effort to gauge the effects of out of order delivery. The hop count field illustrates whether dynamic routing is sufficiently occurring.

For the WAN Routers, the lengths of all queues are examined, both for their absolute occupancy, and for packets on the main conversation.

3.3.4. BONeS Simulation Design

Transfer from an abstract approach into a simulation first requires the construction a BONeS simulation module. Probes are then placed into this module to capture data during the simulation, noting that for all runs the same probe configuration is used (this is done for simplicity). The operation of the Basic Simulation, with details about Parameters and execution script, is given, after which the modifications are described for each subsequent iteration.

Every simulation is run with iteration of the "Global Seed" Parameter, at least three times. This particular aspect is not explicitly outlined because it is carried out so that visual observation can be made to ensure that results are correct. It is fortunate that the automated capability of BONeS allows for this to be carried out quickly and effortlessly.

3.3.4.1. Topology

The approach is translated into an actual BONeS simulation first through the construction of a simulation Module using the components developed on Part 1 of this thesis. The parameters relevant to the simulation are visible in the figure.

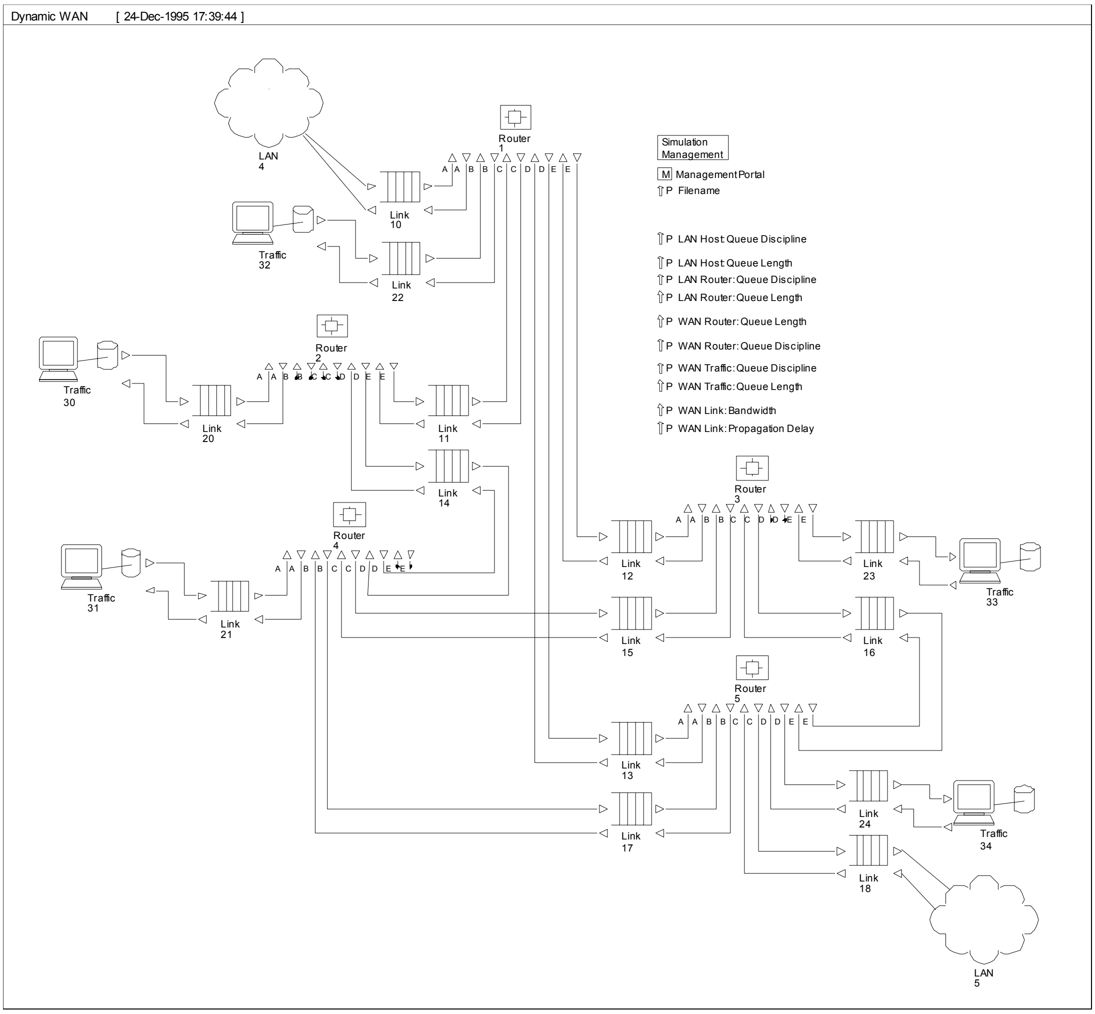


Figure 2-3.7. Simulation Topology: Single TCP Conversation WAN

3.3.4.2. Post Processing and Probe Placement (NOT FINISHED)

ack!

3.3.4.2.1. Basic Simulation

TCP Window Information	
For	Host 11
Purpose	To show the detailed attributes of the TCP congestion control algorithm, as it alters during the course of the simulation. In addition, events that are correlated with TCP congestion control activity are also captured.
X Axis	(Seconds): Time
Y Axis	(Bytes): Congestion Window, Slow Start Threshold, Unacknowledged Data (No Units): Retransmission Events, Timer Expiries
Probes	TCP Probes are used, and they are placed into Host 11's Transport Layer.

TCP Reassembly List Length	
For	Host 21
Purpose	Out of order delivery can be ascertained by looking at the size of the r assembly list, which holds out of order segments.
X Axis	(Seconds): Time
Y Axis	(Integer Value): List Length
Probes	TCP Probes are used, and they are placed into Host 21's Transport Layer.

TCP Computed and Actual Round Trip Time (RTT) Information	
For	Host 11
Purpose	The RTT plays an important role in TCP congestion control. However, as it is estimated, observations of the actual RTT should also be made.
X Axis	(Seconds): Time
Y Axis	(Milliseconds): RTT Value, RTT Value +RTT Variance, RTT Value - RTT Variance, Actual RTT
Probes	TCP Probes are placed into Host 11's Transport Layer. The Actual RTT is obtained by placing a probe into Host 1's Transport Layer to extract the timing information from a received acknowledgement.

WAN Router Queue Information	
For	WAN Routers 1, 2, 3, 4 and 5.
Purpose	The queue drops packets, and affects the RTT for packets. Its behaviour can be correlated with that of TCP congestion control.
X Axis	(Seconds): Time
Y Axis	(Integer) Queue Length, Host Queue Forward Length, Host Reverse Queue Length, Queue Drops
Probes	Queue Probes are placed into each Network Layer through which the conversation between Host 11 and Host 21 passes. The lenght of the queue is captured (in total) along with the length attributed to the conversation in each particular direction.

Transport Layer Data Transmission	
For	Host 11
Purpose	The qualitative information about a conversation is related to its throughput and retransmission levels. The number of transmitted and retransmitted bytes is also affected, and can be correlated with, TCP congestion control activity.
X Axis	(Seconds): Time
Y Axis	(Kilobytes): KB Transmitted, KB Retransmitted
Probes	TCP Probes are placed into Host 11's Transport Layer.

Transport Layer Data Transmission (95% confidence level)	
For	Host 11
Purpose	For greater confidence in the simulation results, a confidence plot using different initial random seeds is used. The information best used on a confidence plot is the throughput and retransmit levels, as the assumption is that they are relevant equivalent for a given scenario. Window and Queue information is more highly variant, and subject to phase differences.
X Axis	(Seconds): Time
Y Axis	(Kilobytes): KB Transmitted, KB Retransmitted
Probes	TCP Probes are placed into Host 11's Transport Layer.

WAN Link Utilisation	
For	All WAN Links
Purpose	Because of retransmission timeouts and other events, the link may not always be fully utilised, where under ideal conditions it should always be.
X Axis	(Seconds): Time
Y Axis	(Percentage): Utilisation
Probes	Probes are placed into all Links through which the conversation between Host 11 and Host 21 passes in the forward direction (i.e. Host 11 -> Host 21). They capture the sum of all packet lengths passed through the link over the total capacity made available by that link according to the length of time in the simulation.

3.3.4.2.2. Queue Length Iteration

When the queue length is iterated, it is expected that various qualitative aspects of the TCP conversation will be affected. The graphs will tend to indicate any correlations.

Throughput versus. Queue Length	
For	Host 11
Purpose	The relationship between Queue Length and Throughput tends to indicate a "good" queue length, and the effects of queuing in general (in a first or second order manner).
X Axis	(Integer Value): Queue Length
Y Axis	(Kilobytes per second): Throughput
Probes	The Probes used are those from the Basic Simulation. The graph is constructed by taking the total number of bytes transmitted for the conversation over the time of the conversation, for each simulation run.

Retransmission Ratio versus. Queue Length	
For	Host 11
Purpose	As the background traffic increases, more retransmissions may occur, due to the loss of acknowledgements or for other reasons.
X Axis	(Integer Value): Queue Length
Y Axis	(Integer Value): Retransmission Ratio
Probes	The Probes used are those from the Basic Simulation. The retransmission ratio is constructed for each conversation in each simulation run.

Average RTT versus. Queue Length	
For	Host 11
Purpose	As Queue Length is increased, the RTT should be noticeably different both in average value and variance.
X Axis	(Integer Value): Queue Length
Y Axis	(Milliseconds): Average RTT Value, Average RTT Variance, Average actual RTT
Probes	The Probes used are those from the Basic Simulation. The graph is constructed by taking the average RTT for the conversation for each simulation run.

3.3.4.2.3. Traffic Level Iteration

Throughput versus. Traffic Level	
For	Host 11
Purpose	As the level of background traffic increases, the throughput levels may suffer due to the share with acknowledgements.
X Axis	(Kilobytes per second): Traffic Level
Y Axis	(Integer Value): Throughput
Probes	The Probes used are those from the Basic Simulation. The RTT ratio is taken by computing the average RTT for both conversations, and dividing them. The throughput is taken as the total number of bytes transmitted for the conversation over the time of the conversation, for each simulation run.

Retransmission Ratio versus. Traffic Level	
For	Host 11
Purpose	As the background traffic increases, more retransmissions may occur, due to the loss of acknowledgements or for other reasons.
X Axis	(Kilobytes per second): Traffic Level
Y Axis	(Integer Value): Retransmission Ratio
Probes	The Probes used are those from the Basic Simulation. The RTT ratio is taken by computing the average RTT for both conversations, and dividing them. The retransmission ratio is constructed for each conversation in each simulation run.

Average RTT versus Traffic Level	
For	Host 1
Purpose	As the background traffic increases, the average RTT should increase due to the increased loading for returned acknowledgments
X Axis	(Kilobytes per second): Traffic Level
Y Axis	(Milliseconds): Average RTT Value, Average RTT Variance, Average actual RTT
Probes	The Probes used are those from the Basic Simulation. The graph is constructed by taking the average RTT for the conversation for each simulation run.

3.3.4.3. Execution: Basic Simulation

In the basic simulation, the Parameters must be configured using the BONeS Set Parameters Dialog. One such parameter is the Management Script. There is no iteration in the basic simulation.

3.3.4.3.1. Parameters

The parameters correspond to the values discussed in the Approach.

Parameter	*Value*	*Description*
Filename	"multipath.txt"	Contains the Management Script..
LAN Host: Queue Discipline	Drop Tail	The LAN does not play a significant part in the simulation, this value is not important.
LAN Host: Queue Length	5	The LAN does not play a significant part in the simulation, this value is not important
LAN Router: Queue Discipline	Drop Tail	The LAN does not play a significant part in the simulation, this value is not important.
LAN Router: Queue Length	5	The LAN does not play a significant part in the simulation, this value is not important
WAN Router: Queue Discipline	Drop Random	For the basic case, choose the best.
WAN Router: Queue Length	8	For the basic case, choose a conservative value.
WAN Traffic: Queue Discipline	Drop Random	For the basic case, choose the best.
WAN Traffic: Queue Length	8	For the basic case, choose a conservative value.
WAN Link: Bandwidth	64kbps	Models an ISDN B Channel
WAN Link: Propagation Delay	100ms	Models a conservative Propagation Delay.

3.3.4.3.2. Management Script

The Management Script is broken up into a number of steps according to the outline given in the Approach. The place at which alterations are made for the Traffic Level iteration is highlighted in bold.

Step 1: Initial configuration at Time 0

Set Routing Entries for Router 1 at the WAN.

```
0 ⇒ Router 1 : Set Route Entry (Addr: Host 41, If: Link 10, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 51, If: Link 11, Cost: 4)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 51, If: Link 12, Cost: 3)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 51, If: Link 13, Cost: 2)

0 ⇒ Router 1 : Set Route Entry (Addr: Traf 32, If: Link 22, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Traf 30, If: Link 11, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Traf 33, If: Link 12, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Traf 34, If: Link 13, Cost: 1)
```

Set Routing Entries for Router 2 at the WAN.

```
0 ⇒ Router 2 : Set Route Entry (Addr: Host 41, If: Link 11, Cost: 2)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 51, If: Link 14, Cost: 3)
```

```
0 ⇒ Router 2 : Set Route Entry (Addr: Traf 30, If: Link 20, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Traf 32, If: Link 11, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Traf 31, If: Link 14, Cost: 1)
```

Set Routing Entries for Router 3 at the WAN -- Note the configuration for Router 4 with regard to Link 15; a potential Loop can occur here.

```
0 ⇒ Router 3 : Set Route Entry (Addr: Host 41, If: Link 12, Cost: 2)
0 ⇒ Router 3 : Set Route Entry (Addr: Host 41, If: Link 15, Cost: 4)
0 ⇒ Router 3 : Set Route Entry (Addr: Host 51, If: Link 15, Cost: 3)
0 ⇒ Router 3 : Set Route Entry (Addr: Host 51, If: Link 16, Cost: 2)

0 ⇒ Router 3 : Set Route Entry (Addr: Traf 33, If: Link 23, Cost: 1)
0 ⇒ Router 3 : Set Route Entry (Addr: Traf 32, If: Link 12, Cost: 1)
0 ⇒ Router 3 : Set Route Entry (Addr: Traf 31, If: Link 15, Cost: 1)
0 ⇒ Router 3 : Set Route Entry (Addr: Traf 34, If: Link 16, Cost: 1)
```

Set Routing Entries for Router 4 at the WAN -- Note the configuration for Router 3 with regard to Link 15, a potential Loop can occur here.

```
0 ⇒ Router 4 : Set Route Entry (Addr: Host 41, If: Link 14, Cost: 3)
0 ⇒ Router 4 : Set Route Entry (Addr: Host 41, If: Link 15, Cost: 3)
0 ⇒ Router 4 : Set Route Entry (Addr: Host 51, If: Link 15, Cost: 3)
0 ⇒ Router 4 : Set Route Entry (Addr: Host 51, If: Link 17, Cost: 2)

0 ⇒ Router 4 : Set Route Entry (Addr: Traf 31, If: Link 21, Cost: 1)
0 ⇒ Router 4 : Set Route Entry (Addr: Traf 30, If: Link 14, Cost: 1)
0 ⇒ Router 4 : Set Route Entry (Addr: Traf 33, If: Link 15, Cost: 1)
0 ⇒ Router 4 : Set Route Entry (Addr: Traf 34, If: Link 17, Cost: 1)
```

Set Routing Entries for Router 5 at the WAN.

```
0 ⇒ Router 5 : Set Route Entry (Addr: Host 41, If: Link 13, Cost: 2)
0 ⇒ Router 5 : Set Route Entry (Addr: Host 41, If: Link 16, Cost: 3)
0 ⇒ Router 5 : Set Route Entry (Addr: Host 41, If: Link 17, Cost: 4)
0 ⇒ Router 5 : Set Route Entry (Addr: Host 51, If: Link 18, Cost: 1)

0 ⇒ Router 5 : Set Route Entry (Addr: Traf 34, If: Link 24, Cost: 1)
0 ⇒ Router 5 : Set Route Entry (Addr: Traf 32, If: Link 13, Cost: 1)
0 ⇒ Router 5 : Set Route Entry (Addr: Traf 33, If: Link 16, Cost: 1)
0 ⇒ Router 5 : Set Route Entry (Addr: Traf 31, If: Link 17, Cost: 1)
```

Set Routing Entries for the Router in LAN 4.

```
0 ⇒ Router 40 : Set Route Entry (Addr: Host 41, If: Link 46, Cost: 1)
0 ⇒ Router 40 : Set Route Entry (Addr: Host 51, If: Link 10, Cost: 1)
```

Set Routing Entries for the Router in LAN 5.

```
0 ⇒ Router 50 : Set Route Entry (Addr: Host 51, If: Link 56, Cost: 1)
0 ⇒ Router 50 : Set Route Entry (Addr: Host 41, If: Link 18, Cost: 1)
```

Step 2: Establishment of TCP conversation between Host 41 in LAN 4 and Host 51 in LAN 5 at Time 0

Set Initial Sequence Numbers for Host 41 and Host 51.

```
0 ⇒ Host 41 : Set Parameters (ISN: 12345678)
0 ⇒ Host 51 : Set Parameters (ISN: 12345678)
```

Request Host 41 to Connect Session to Host 51, and Host 51 to Connect Session to Host 41.

```
0 ⇒ Host 41 : Connect Session (Addr: Host 51)
0 ⇒ Host 51 : Connect Session (Addr: Host 41)
```

Instruct the Generator on Host 41 to produce a single Constant unit of data.

```
0 ⇒ Host 41 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value:
10000000))
```

Step 3: Establishment of Jitterisation Traffic at Time 60

Set Address Lists on Traffic 30, 31, 32, 33 and 34.

```
60 ⇒ Host 30 : Set Address List (Num: 2, Addr: 31/32)
60 ⇒ Host 31 : Set Address List (Num: 3, Addr: 30/33/34)
60 ⇒ Host 32 : Set Address List (Num: 3, Addr: 30/33/34)
60 ⇒ Host 33 : Set Address List (Num: 3, Addr: 32/31/34)
60 ⇒ Host 34 : Set Address List (Num: 3, Addr: 31/32/33)
```

Instruct the Generator on Traffic 30, 31, 32, 33 and 34 to produce Poisson units of data -- these values are equivalent, but under iteration, they all proceed for 120 seconds.

```
60 ⇒ Host 30 : Setup Statistical Generator (Time: 120, Bytes: 0, Count: 0, Time (Type: POISSON, Lambda: X), Space (Type: CONSTANT, Value: <ITER>))
60 ⇒ Host 31 : Setup Statistical Generator (Time: 120, Bytes: 0, Count: 0, Time (Type: POISSON, Lambda: X), Space (Type: CONSTANT, Value: <ITER>))
60 ⇒ Host 32 : Setup Statistical Generator (Time: 120, Bytes: 0, Count: 0, Time (Type: POISSON, Lambda: X), Space (Type: CONSTANT, Value: <ITER>))
60 ⇒ Host 33 : Setup Statistical Generator (Time: 120, Bytes: 0, Count: 0, Time (Type: POISSON, Lambda: X), Space (Type: CONSTANT, Value: <ITER>))
60 ⇒ Host 34 : Setup Statistical Generator (Time: 120, Bytes: 0, Count: 0, Time (Type: POISSON, Lambda: X), Space (Type: CONSTANT, Value: <ITER>))
```

Step 4: Terminate the simulation at Time 180

Stop.

```
180 ⇒ :
```

The Management script is constructed by translating these pseudo operations using the information provided in Part 1. This is not provided here, as it is cryptic and pointless.

3.3.4.4. Execution: Queue Length Iteration Simulation

The same Parameters and Management script are used as in the Basic Simulation, however for the WAN Queue Length Parameter a BONeS iteration dialog is selected. This dialog is instructed to step through the WAN Queue Length from values 1 to 64 inclusive.

3.3.4.5. Execution: Traffic Level Iteration Simulation

The same Parameters are used as in the Basic Simulation, however a number of Management Scripts are created, to iterate the "Space" Parameter for the Statistical Generator between 1 and 256.

3.3.5. Expectations (NOT FINISHED)

--summary--

the simulation starts with the tcp covnerstaion between host 21 and host 22. the behaviour exhibited will be similar to that of previous conversions. there will be a number of differences. it is expected that there will be an increase in out of order deliveries, these can be seen through the reassmbly list size. it is also expected that the round trip times will be larger and more variant due to the multiple paths. in fact, we should see some defined levels at the path lenghts. we can look at the hop count and see that packets have taken longer paths through the network. the dynamic routing is such that packets are placed onto interfaces that have lower loads. what this means is that the load is distribruted throughout the queeus in a single router. so this means that for bursty traffic, there is more queue space to accomodate it. we expect to see ack compresssion occur, it was shown that this results in bursty traffic. generally if the queue has data in it, the bursty traffic is evened out. it may therefore be evened out by the front end router, but distributed. during the transient slow start the tcp will attempt to reach a stable network operating point. previously, we could ocmpute this, but now it will be harder. the point will be the sum of the paths. but there are other factors. the ack compression problem will reduce out of order delivery will cause problems. it is expected to occur. and as a result it is expected to see increased retransmit levels. these will affect the slow start threshold. they will also impact on the throughput and retransmission ratio. we want to observe whether the performance is fairly predictable, or whether it is erratic. we want to observe whether or not the basic congestion control behaviour is there, we actually are not sure at the moment whether it will or will not be.

when we look at the increase in quue levels, we sohuld see increase in rtts. the increased rtts may cause more retranmssions. they may also exacerbate the out of order delivery problem. we have found that with greater qeueue levels

when we look at the increase in traffic levels. we should see results similar to lower queue levels. we should also see ackcompression as well, probably more distinct than it was otherwise.

conclusion; points to the fact that dynamic networks do impact. does tend to show that current simulations may need to address these types of things

*

--text--

the simulation commences with the TCP conversation between host 21 and host 22. in general, we expect that the behaviour of the TCP conversation is close to that seen in the previous simulations. This means that it will iniitally consist of a large slow start threshold, and with the rapid increase in the congestion window will tend to congestion the network in a short time. the significant difference in this simulation is that the network consists of multiple paths between the transmitter and the receiver.

the routing is constructed in such a manner that output interfaces are slected based on destination address and on the loading levels at the queue. as the window increases, its data will tend to be distributed throughout the network. because rather than the queue dropping a packet, it will be sent down other path. in some respects, this is equivalent to it suffering an additional delay. in previous simulations, we could predict the point at which congestion occurred by consider the total space available in the network.

for this simulation, the prediction fo the congsetion window is not so easy. firstly, the window can be expected to be the sum of the paths between the transmitter and receiver, but there are further complications. the first complication is that acknowledgements will also be subject to dynamic routing, and therefore acknowledgement compression will result. as shown in the previous simulation, this causes a bursty nature in the tcp transmitter, and this bursty nature would tend to results in earlier congestion -- i.e. congestion wll occur high up in the network. if this does occur, we will see it through queue drops being more prevalent higher up in the network.

another added complexity is that with the dynamic routing, we can expect to see out of order delivery occur. this results from the fact that as packets travel down different paths, they suffer different delays. with out of order delivery, the level of retransmissions may increase due to duplicate acknowledgements being misinterpreted as retransmission timeouts. byt the same token, the the duplicate acknowledgement and “fast recovery” procedure will reduce the slow start and congestion windows.

we can determine whether or not out of order delivery is responsible for retransmissions by looking at the size of the reassembly queue, and whether or not a sgement was actually dropped in the entwork, or whether ir arrives a short time later.

by now, it is fairly obvious that clear picture of the congestion window is not clear at the moment.

as the conversation proceeds through the cyclic probing phase of congestion avoidance, we do expect

xxx

the conversation starts. it goes through the slow start mechanisms. it reaches congestion. but we don’t know where it reaches congestion. we know that due to the nature of hte network it will be the sum of all the links.

- we want to see whether or not this increases to a large value, due to drops in the network, or due to out of order delivery

- then we want to look and see whether or not the dupacks fired the retransmit at the transmitter

- we should observe some big variations in the rtt for the connection due to the paths

- if we look at this enough, the rtt should tend to exist at defined levels that corespond with all the link bandwidth delays

- the effect though, will be that we will see the case of ack compression

- we expect that if ack compression occurs, then we should see bursts of packets

- we want to look and see whether or not these bursts can be picked up at the start of the network

- the impact on the tcp window information is that the rtt variations means things won’t be nice and linear

- as the connection progresses, the cyclic nature should still be there

- we expect to see that dynamic routing will occur because of the conversation itself

- we can check that this does occur by observing the queue lengths in the networks
- but we can also look at the hop count in received packets
- as for the throughput of the conversation, we expect it to be fairly constant

(queue level iteration)

- we are interested in the case of the iterated queue levels in the network
- what we expect to see is firstly an increase in the rtt through the network
- however, the potential impact of increased rtt is on the level of retransmissions
- and the problems with out of order delivery
- this is expected because the more rtt we have, the more space we have in the network
- when we consider throughput against rtt, we should see that it actually goes down
- the reason it goes down though is because of the retransmit levels and out of order delivery

(traffic level iteration)

- the introduction of traffic will increase loading and losses in the network
- it also causes ack compression
- what we are interested in looking is how valid our observations are with other background traffic

3.3.6. Execution of Simulation

The simulation was not executed due to the problems surrounding the unavailability of the BONeS software.

3.3.7. Analysis of Results

No results were gathered from the simulation due to the problems surrounding the unavailability of the BONeS software.

3.3.8. Conclusions (NOT FINISHED)

we wanted to examine the effects of dynamic routing and complex network topologies. although no simulations were carried out, our expectations point to decreased levels of performance. the result of these complex networks will tend to be increased levels of retransmissions due to increased levels of out or order delivery causing the tcp fast retransmit to inadvertantly fire. the same mechanism will reduce the slow start threshold and congestion window, further decreasing performance. providing more queueing the network is not expected to help either, only helping to increase the incidence of out of order delivery, and increase levels of retransmissions, whilst decreasing throughput. As there is interchangability between queueing levels, bandwidth and delay, this tends to indicate that central problems are out of order delivery and widely variant RTTs. providing traffic in the network, as is expected in a realistic situation, will only further exacerbate the issue, with subsequent increase in losses and fall in throughput.

in summary, this environment does provide decreased levels of performance, especially when considered in light of the previous simulations.

3.4. Multiple TCP conversations overloading long-haul WAN Link

3.4.1. Problem and Objectives

This scenario concerns itself with the second major issue identified in relation to the Transmission Control Protocol's (TCP) congestion control mechanisms as they apply in Wide-Area Network (WAN) environments.

The concern is based upon the knowledge that increasing traffic levels are being experienced by central backbone links that interconnect large WANs. Particular situations can arise that the TCP congestion control mechanisms may not be capable of servicing due to fundamental limitations.

Central backbone links must support a huge number of conversations, [Ref] reports that it is usual to see some 400 simultaneous TCP conversations on the main Internet link between the US and Europe. All of these conversations must share the medium appropriately, and therefore would tend to receive a small portion of the available bandwidth and queuing space.

The TCP is a window-based protocol, and when restricted by the congestion window will send a minimum of two bytes into the network during each Round Trip Time (RTT), based upon the reception of acknowledgements from its receiver. The implicit assumption here is that the network is capable of supporting, at a minimum, a two-byte window for each conversation. On heavily overloaded links, with a large number of conversations, this assumption can be invalid.

This problem is referred to as the "window-granularity" problem.

Therefore, *the objective in this scenario is to examine the window-granularity problem by generating the suspected conditions, and observing the effects that result.*

3.4.2. Discussion and Related Work (NOT FINISHED)

ack!

3.4.3. Approach

The approach consists of identifying the model, simulation and observations that are required to obtain the objectives.

Model

The model consists of a number of LANs bridged by a single WAN Link. The intent of the model is to capture a general case where a WAN Link is required to carry many conversations. Each "side" of the WAN Link has four LANs, of which there are two Hosts in each.

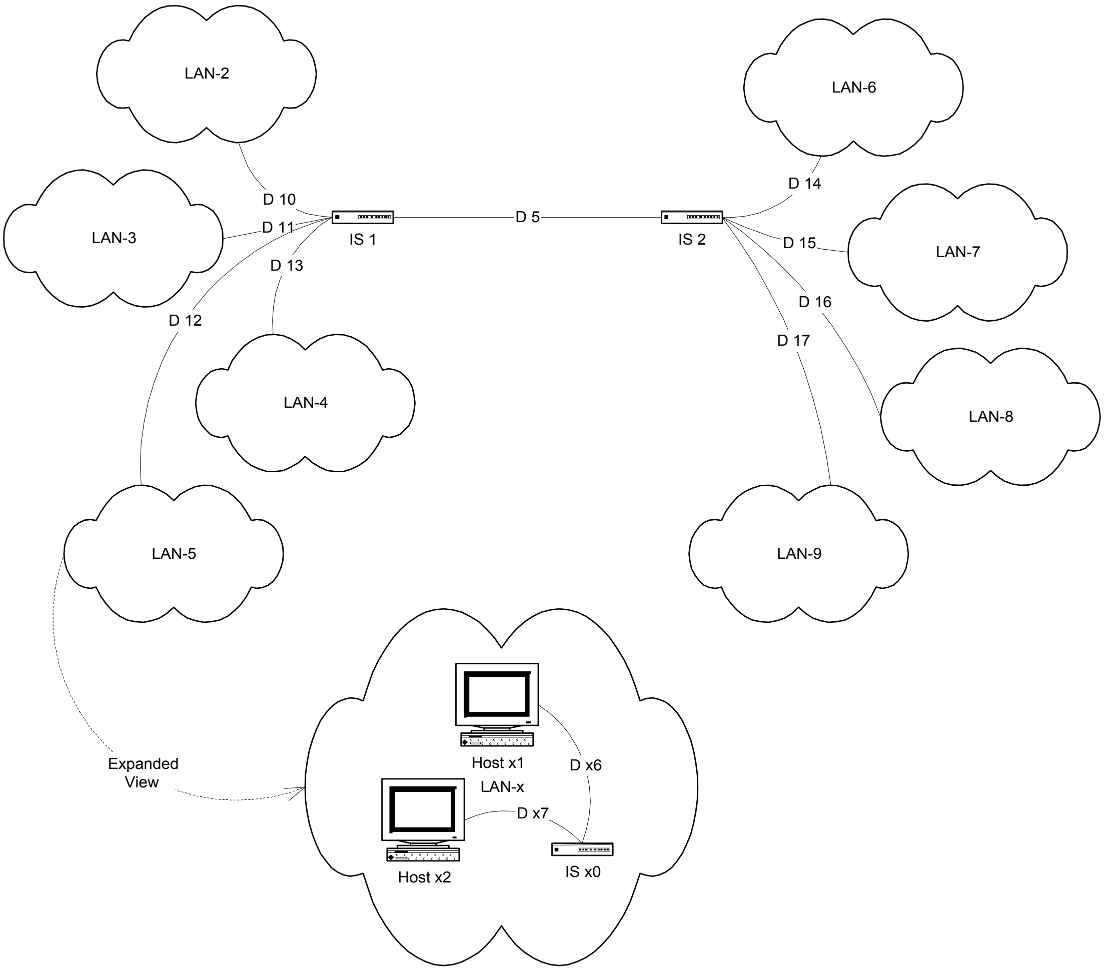


Figure 2-3.8. Simulation Model: Multiple TCP Conversation L-WAN

Most parameters in the simulation are fixed. The LANs have Links set with Bandwidth and Propagation Delays corresponding typical Ethernet networks, i.e. 10Mbps and 1ms respectively. The LAN Router queue lengths and disciplines are set to 16 and Drop Tail respectively; they are not expected to play an operational role in congestion. Each LAN is connected to a WAN Router through a WAN Link, which has Bandwidth and Propagation Delay set to 64Kbps and 20ms respectively. This models a conservative WAN. The WAN Routers request Queue Length and Queue Discipline parameters, these are set to 8 and Drop Tail respectively, however are subject to iteration during simulation. The WAN Link is set to have a Bandwidth of 16Kbps and a Propagation Delay of 20ms.

The parameters must be set so that congestion occurs in the WAN Router due to the WAN Link; at the same time, the WAN Link's Bandwidth and Propagation Delay are important as the relationship between them and the TCP conversation is central to the problem at hand.

Basic Simulation

The basic simulation consists of many TCP conversations established from Hosts in LAN 2, LAN 3, LAN 4 and LAN 5 to LAN 6, LAN 7, LAN 8 and LAN 9 respectively. The conversation carries one-way traffic (a transfer of a large unit of data) in the forward direction, however the return traffic consists only of acknowledgements. These conversations run for 120 seconds.

During this phase, the only phase, of the simulation, a few key characteristics of all TCP transmitters are of interest. The queue length for the WAN Router for the WAN Link is where congestion should be seen, and full utilisation of the WAN Link should also be apparent.

The simulation is stopped after 120 seconds.

Variations

There are two variations of concern. The first involves the WAN Router where it is desired to examine the effects of altered Queue lengths. Therefore, simulations are run with Queue lengths between 1 and 64 (packets).

The second involves the WAN Link. It is desired to examine the particular overload problem, as it tends to become worse, for this a reduction in the total space within the network must be carried out. It is done by iterating on the Bandwidth (we can actually alter either the Bandwidth or Propagation Delay, there seems to be little difference, however by altering the Propagation Delay, we are moving more of the queue into the network, so to speak).

Observations

The observations are gained from two sources. The TCP transmitters, all of them, provide information about their congestion control characteristics and more general items such as retransmission levels. From these, the case can be seen where the congestion window lowers and at the same time the level of retransmissions increase. Other fine-grained TCP information is not particularly important.

The WAN Router provides important information about its Queue lengths, and the extent to which it drops items from the Queue. From this, congestion effects can be examined.

3.4.4. BONeS Simulation Design

Transfer from an abstract approach into a simulation first requires the construction a BONeS simulation module. Probes are then placed into this module to capture data during the simulation, noting that for all runs the same probe configuration is used (this is done for simplicity). The operation of the Basic Simulation, with details about Parameters and execution script, is given, after which the modifications are described for each subsequent iteration.

Every simulation is run with iteration of the “Global Seed” Parameter, at least three times. This particular aspect is not explicitly outlined because it is carried out so that visual observation can be made to ensure that results are correct. It is fortunate that the automated capability of BONeS allows for this to be carried out quickly and effortlessly.

3.4.4.1. Topology

The approach is translated into an actual BONeS simulation first through the construction of a simulation Module using the components developed on Part 1 of this thesis. The parameters relevant to the simulation are visible in the figure.

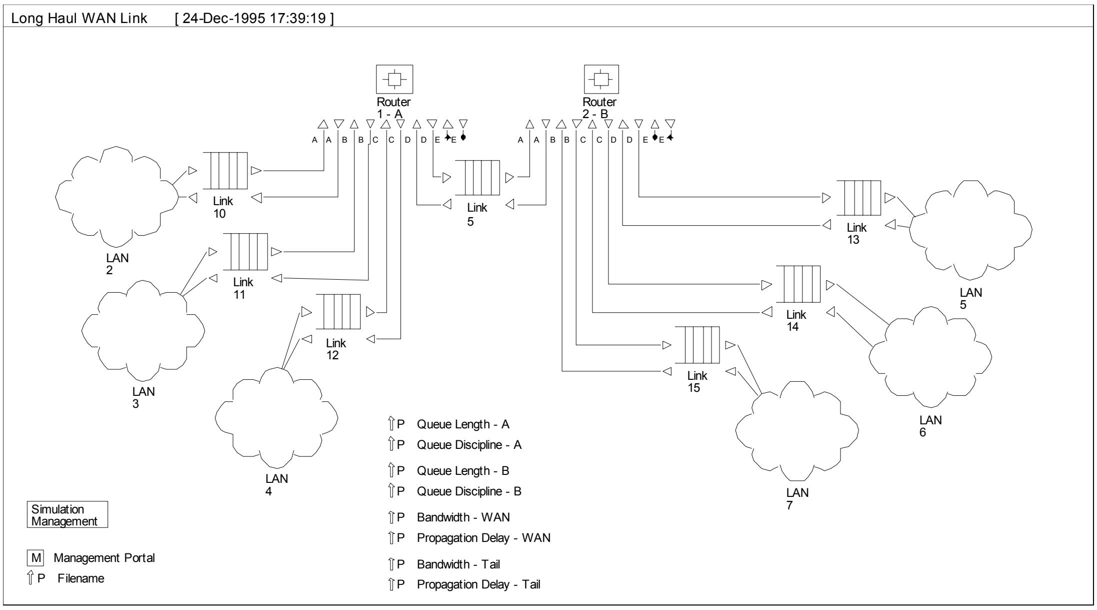


Figure 2-3.9. Simulation Topology: Multiple TCP Conversation L-WAN

3.4.4.2. Post Processing and Probe Placement

To construct information used in the analysis, Probes can be placed into the simulation using the BONeS Simulation Manager; once placed, they are then used in the Post Processor to generate graphs. The approach taken here is to first identify the particular graphs that indicate critical information for analysis, and then to determine which Probes must be placed, and where they must be placed.

3.4.4.2.1. Basic Simulation

For the basic simulation, the graphs illustrate the lifecycle activity in the host and the network.

TCP Window Information	
For	All Hosts
Purpose	Only central TCP congestion window information is required, for all hosts and as an average.
X Axis	(Seconds): Time
Y Axis	(Bytes): Congestion Window, Slow Start Threshold, Average Congestion Window (95% confidence level), Average Slow Start Threshold (95% confidence level) (No Units): Retransmission Events, Average Retransmission Events (95% confidence level)
Probes	TCP Probes are used, and they are placed into each Host's Transport Layer. The averages are computed from all the values, and a 95% confidence level is also shown. The averages will potentially only be useful once steady state has been reached.

TCP Computed and Actual Round Trip Time (RTT) Information	
For	All Hosts
Purpose	The RTT plays an important role in TCP congestion control. However, as it is estimated, observations of the actual RTT should also be made.
X Axis	(Seconds): Time
Y Axis	(Milliseconds): RTT Value, RTT Value +RTT Variance, RTT Value - RTT Variance, Actual RTT, Average RTT Value (95% confidence level), Average Actual RTT (95% confidence level)
Probes	TCP Probes are placed into each Host's Transport Layer. The Actual RTT is obtained by placing a Probe into the Host's Transport Layer to extract the timing information from a received acknowledgement. The averages may take a while to settle.

WAN Router Queue Information	
For	WAN Router 1
Purpose	The WAN Router is the central bottleneck
X Axis	(Seconds): Time
Y Axis	(Integer Value): Queue Length, Queue Drops
Probes	Queue Probes are placed into Wan Router 1's Network Layer leading to the WAN Link.

Transport Layer Data Transmission	
For	All
Purpose	The qualitative information about a conversation is related to its throughput and retransmission levels. The number of transmitted and retransmitted bytes is also affected, and can be correlated with, TCP congestion control activity.
X Axis	(Seconds): Time
Y Axis	(Kilobytes): KB Transmitted, KB Retransmitted, Average KB Transmitted (95% confidence), Average KB Retransmitted (95% confidence).
Probes	TCP Probes are placed into each Host's Transport Layer.

WAN Link Utilisation	
For	WAN Link
Purpose	Because of retransmission timeouts and other events, the link may not always be fully utilised, where under ideal conditions it should always be.
X Axis	(Seconds): Time
Y Axis	(Percentage): Utilisation
Probes	Probes are placed into the WAN Link. They capture the sum of all packet lengths passed through the link over the total capacity made available by that link according to the length of time in the simulation.

3.4.4.2.2. Queue Length Iteration

When the Queue Length is iterated, the averages obtained through the Basic Simulation are of interest. Many of these values compute averages of average information seen in the Basic Simulation.

Average Average Congestion Window versus. Queue Length	
For	All
Purpose	The average congestion window indicates the amount of data that the host thinks that the network can support.
X Axis	(Integer Value): Queue Length
Y Axis	(Bytes): Average Average Congestion Window
Probes	The Probes used are those from the Basic Simulation. The average of the Average Congestion window seen is evaluated.

Average Average RTT versus. Queue Length	
For	All
Purpose	As Queue Length is increased, the RTT should be noticeably different both in average value and variance.
X Axis	(Integer Value): Queue Length
Y Axis	(Milliseconds): Average Average RTT Value, Average Average RTT Variance, Average Average Actual RTT
Probes	The Probes used are those from the Basic Simulation. The graph is constructed by taking the average RTT computed for all the conversations for each simulation run. The average is computed of the Averages from the basic simulation.

Average Average Throughput versus. Queue Length	
For	All
Purpose	The relationship between Queue Length and Throughput tends to indicate a "good" queue length, and the effects of queuing in general (in a first or second order manner).
X Axis	(Integer Value): Queue Length
Y Axis	(Kilobytes per second): Average Average Throughput
Probes	The Probes used are those from the Basic Simulation. The averages are used from each simulation by taking the average data transmitted over the time for the simulation.

Average Retransmission Ratio versus. Queue Length	
For	All
Purpose	Queue Lengths and Retransmission Ratios may be correlated. The retransmission ratio is determined by taking the total number of retransmitted bytes for a conversation and dividing by the total number of transmitted bytes.
X Axis	(Integer Value): Queue Length
Y Axis	(Integer): Average Retransmission Ratio
Probes	The Probes used are those from the Basic Simulation. The averages are used from each simulation by taking the average data retransmitted over the time for the simulation.

3.4.4.2.3. WAN Bandwidth Iteration

As the WAN Bandwidth is altered, the same graphs computed for the Queue Length iteration are used, however the Bandwidth * Delay Product is used on the X Axis.

3.4.4.3. Execution: Basic Simulation

In the basic simulation, the Parameters must be configured using the BONeS Set Parameters Dialog. One such parameter is the Management Script. There is no iteration in the basic simulation.

3.4.4.3.1. Parameters

The parameters correspond to the values discussed in the Approach.

Parameter	*Value*	*Description*
Filename	"overload.txt"	Contains the Management Script
Router 1: Queue Discipline	Random Drop	The discipline is not strictly important, so choose the best.
Router 1: Queue Length	4	Choose an arbitrary value, investigate others in iterations.
Router 2: Queue Discipline	Random Drop	The discipline is not strictly important, so choose the best.
Router 2: Queue Length	4	Choose an arbitrary value, investigate others in iterations.
WAN Link: Bandwidth	16kbps	The Bandwidth needs to be small, possibly this will have to be tailored
WAN Link: Propagation Delay	100ms	The Propagation Delay needs to be small, possibly this will have to be tailored.
Tail Link: Bandwidth	64kbps	Not expressly important, so Module an ISDN B Channel.
Tail Link: Propagation Delay	20ms	Not expressly important, so choose conservative value.

3.4.4.3.2. Management Script

The Management Script is broken up into a number of steps according to the outline given in the Approach.

Step 1: Initial configuration at Time 0

Set Routing Entries for Router 1 at the WAN -- The entries are such that all Hosts on the left hand side of the WAN Link are visible through specific Links, whereas all Hosts on the right hand side are visible through the common WAN Link.

```
0 ⇒ Router 1 : Set Route Entry (Addr: Host 21, If: Link 10, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 22, If: Link 10, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 23, If: Link 10, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 24, If: Link 10, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 31, If: Link 11, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 32, If: Link 11, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 33, If: Link 11, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 34, If: Link 11, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 41, If: Link 12, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 42, If: Link 12, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 43, If: Link 12, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 44, If: Link 12, Cost: 1)
```

```
0 ⇒ Router 1 : Set Route Entry (Addr: Host 51, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 52, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 53, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 54, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 61, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 62, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 63, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 64, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 71, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 72, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 73, If: Link 5, Cost: 1)
0 ⇒ Router 1 : Set Route Entry (Addr: Host 74, If: Link 5, Cost: 1)
```

Set Routing Entries for Router 2 at the WAN -- The entries are such that all Hosts on the right hand side of the WAN Link are visible through specific Links, whereas all Hosts on the left hand side are visible through the common WAN Link.

```
0 ⇒ Router 2 : Set Route Entry (Addr: Host 51, If: Link 13, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 52, If: Link 13, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 53, If: Link 13, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 54, If: Link 13, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 61, If: Link 14, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 62, If: Link 14, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 63, If: Link 14, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 64, If: Link 14, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 71, If: Link 15, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 72, If: Link 15, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 73, If: Link 15, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 74, If: Link 15, Cost: 1)

0 ⇒ Router 2 : Set Route Entry (Addr: Host 21, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 22, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 23, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 24, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 31, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 32, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 33, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 34, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 41, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 42, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 43, If: Link 5, Cost: 1)
0 ⇒ Router 2 : Set Route Entry (Addr: Host 44, If: Link 5, Cost: 1)
```

Set Routing Entries for the Router in LAN 2 -- Hosts in LAN 2 only ever communicate with Hosts in LAN 5.

```
0 ⇒ Router 20 : Set Route Entry (Addr: Host 21, If: Link 26, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 22, If: Link 27, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 23, If: Link 28, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 24, If: Link 29, Cost: 1)

0 ⇒ Router 20 : Set Route Entry (Addr: Host 51, If: Link 10, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 52, If: Link 10, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 53, If: Link 10, Cost: 1)
0 ⇒ Router 20 : Set Route Entry (Addr: Host 54, If: Link 10, Cost: 1)
```

Set Routing Entries for the Router in LAN 3 -- Hosts in LAN 3 only ever communicate with Hosts in LAN 6.

```
0 ⇒ Router 30 : Set Route Entry (Addr: Host 31, If: Link 36, Cost: 1)
0 ⇒ Router 30 : Set Route Entry (Addr: Host 32, If: Link 37, Cost: 1)
0 ⇒ Router 30 : Set Route Entry (Addr: Host 33, If: Link 38, Cost: 1)
0 ⇒ Router 30 : Set Route Entry (Addr: Host 34, If: Link 39, Cost: 1)

0 ⇒ Router 30 : Set Route Entry (Addr: Host 61, If: Link 11, Cost: 1)
0 ⇒ Router 30 : Set Route Entry (Addr: Host 62, If: Link 11, Cost: 1)
```

```
0 ⇒ Router 30 : Set Route Entry (Addr: Host 63, If: Link 11, Cost: 1)
0 ⇒ Router 30 : Set Route Entry (Addr: Host 64, If: Link 11, Cost: 1)
```

Set Routing Entries for the Router in LAN 4 -- Hosts in LAN 4 only ever communicate with Hosts in LAN 7.

```
0 ⇒ Router 40 : Set Route Entry (Addr: Host 41, If: Link 46, Cost: 1)
0 ⇒ Router 40 : Set Route Entry (Addr: Host 42, If: Link 47, Cost: 1)
0 ⇒ Router 40 : Set Route Entry (Addr: Host 43, If: Link 48, Cost: 1)
0 ⇒ Router 40 : Set Route Entry (Addr: Host 44, If: Link 49, Cost: 1)

0 ⇒ Router 40 : Set Route Entry (Addr: Host 71, If: Link 12, Cost: 1)
0 ⇒ Router 40 : Set Route Entry (Addr: Host 72, If: Link 12, Cost: 1)
0 ⇒ Router 40 : Set Route Entry (Addr: Host 73, If: Link 12, Cost: 1)
0 ⇒ Router 40 : Set Route Entry (Addr: Host 74, If: Link 12, Cost: 1)
```

Set Routing Entries for the Router in LAN 5 -- Hosts in LAN 5 only ever communicate with Hosts in LAN 2.

```
0 ⇒ Router 50 : Set Route Entry (Addr: Host 51, If: Link 56, Cost: 1)
0 ⇒ Router 50 : Set Route Entry (Addr: Host 52, If: Link 57, Cost: 1)
0 ⇒ Router 50 : Set Route Entry (Addr: Host 53, If: Link 58, Cost: 1)
0 ⇒ Router 50 : Set Route Entry (Addr: Host 54, If: Link 59, Cost: 1)

0 ⇒ Router 50 : Set Route Entry (Addr: Host 21, If: Link 13, Cost: 1)
0 ⇒ Router 50 : Set Route Entry (Addr: Host 22, If: Link 13, Cost: 1)
0 ⇒ Router 50 : Set Route Entry (Addr: Host 23, If: Link 13, Cost: 1)
0 ⇒ Router 50 : Set Route Entry (Addr: Host 24, If: Link 13, Cost: 1)
```

Set Routing Entries for the Router in LAN 6 -- Hosts in LAN 6 only ever communicate with Hosts in LAN 3.

```
0 ⇒ Router 60 : Set Route Entry (Addr: Host 61, If: Link 66, Cost: 1)
0 ⇒ Router 60 : Set Route Entry (Addr: Host 62, If: Link 67, Cost: 1)
0 ⇒ Router 60 : Set Route Entry (Addr: Host 63, If: Link 68, Cost: 1)
0 ⇒ Router 60 : Set Route Entry (Addr: Host 64, If: Link 69, Cost: 1)

0 ⇒ Router 60 : Set Route Entry (Addr: Host 31, If: Link 14, Cost: 1)
0 ⇒ Router 60 : Set Route Entry (Addr: Host 32, If: Link 14, Cost: 1)
0 ⇒ Router 60 : Set Route Entry (Addr: Host 33, If: Link 14, Cost: 1)
0 ⇒ Router 60 : Set Route Entry (Addr: Host 34, If: Link 14, Cost: 1)
```

Set Routing Entries for the Router in LAN 7 -- Hosts in LAN 7 only ever communicate with Hosts in LAN 4.

```
0 ⇒ Router 70 : Set Route Entry (Addr: Host 71, If: Link 76, Cost: 1)
0 ⇒ Router 70 : Set Route Entry (Addr: Host 72, If: Link 77, Cost: 1)
0 ⇒ Router 70 : Set Route Entry (Addr: Host 73, If: Link 78, Cost: 1)
0 ⇒ Router 70 : Set Route Entry (Addr: Host 74, If: Link 79, Cost: 1)

0 ⇒ Router 70 : Set Route Entry (Addr: Host 41, If: Link 15, Cost: 1)
0 ⇒ Router 70 : Set Route Entry (Addr: Host 42, If: Link 15, Cost: 1)
0 ⇒ Router 70 : Set Route Entry (Addr: Host 43, If: Link 15, Cost: 1)
0 ⇒ Router 70 : Set Route Entry (Addr: Host 44, If: Link 15, Cost: 1)
```

Step 2: Establishment of TCP conversation between Hosts in LAN 2 and Hosts in LAN 5

Set Initial Sequence Numbers for Hosts 21, 22, 23 and 24 and Hosts 51, 52, 53 and 54.

```
0 ⇒ Host 21 : Set Parameters (ISN: 12345678)
0 ⇒ Host 51 : Set Parameters (ISN: 12345678)

0 ⇒ Host 22 : Set Parameters (ISN: 12345678)
0 ⇒ Host 52 : Set Parameters (ISN: 12345678)

0 ⇒ Host 23 : Set Parameters (ISN: 12345678)
0 ⇒ Host 53 : Set Parameters (ISN: 12345678)
```

```
0 ⇒ Host 24 : Set Parameters (ISN: 12345678)
0 ⇒ Host 54 : Set Parameters (ISN: 12345678)
```

Request Hosts 21, 22, 23 and 24 to Connect Session to Hosts 51, 52, 53 and 54 (respectively) and Hosts 51, 52, 53 and 54 to Connect Session to Hosts 21, 22, 23 and 24 (respectively).

```
0 ⇒ Host 21 : Connect Session (Addr: Host 51)
0 ⇒ Host 51 : Connect Session (Addr: Host 21)

0 ⇒ Host 22 : Connect Session (Addr: Host 52)
0 ⇒ Host 52 : Connect Session (Addr: Host 22)

0 ⇒ Host 23 : Connect Session (Addr: Host 53)
0 ⇒ Host 53 : Connect Session (Addr: Host 23)

0 ⇒ Host 24 : Connect Session (Addr: Host 54)
0 ⇒ Host 54 : Connect Session (Addr: Host 24)
```

Instruct the Generator on Hosts 21, 22, 23 and 24 to produce a single Constant unit of data.

```
0 ⇒ Host 21 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value:
10000000))

0 ⇒ Host 22 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value:
10000000))

0 ⇒ Host 23 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value:
10000000))

0 ⇒ Host 24 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value:
10000000))
```

Step 3: Establishment of TCP conversation between Hosts in LAN 3 and Hosts in LAN 6.

Set Initial Sequence Numbers for Hosts 31, 32, 33 and 34 and Hosts 61, 62, 63 and 64.

```
0 ⇒ Host 31 : Set Parameters (ISN: 12345678)
0 ⇒ Host 61 : Set Parameters (ISN: 12345678)

0 ⇒ Host 32 : Set Parameters (ISN: 12345678)
0 ⇒ Host 62 : Set Parameters (ISN: 12345678)

0 ⇒ Host 33 : Set Parameters (ISN: 12345678)
0 ⇒ Host 63 : Set Parameters (ISN: 12345678)

0 ⇒ Host 34 : Set Parameters (ISN: 12345678)
0 ⇒ Host 64 : Set Parameters (ISN: 12345678)
```

Request Hosts 31, 32, 33 and 34 to Connect Session to Hosts 61, 62, 63 and 64 (respectively) and Hosts 61, 62, 63 and 64 to Connect Session to Hosts 31, 32, 33 and 34 (respectively).

```
0 ⇒ Host 31 : Connect Session (Addr: Host 61)
0 ⇒ Host 61 : Connect Session (Addr: Host 31)

0 ⇒ Host 32 : Connect Session (Addr: Host 62)
0 ⇒ Host 62 : Connect Session (Addr: Host 32)

0 ⇒ Host 33 : Connect Session (Addr: Host 63)
0 ⇒ Host 63 : Connect Session (Addr: Host 33)
```

```
0 ⇒ Host 34 : Connect Session (Addr: Host 64)
0 ⇒ Host 64 : Connect Session (Addr: Host 34)
```

Instruct the Generator on Hosts 31, 32, 33 and 34 to produce a single Constant unit of data.

```
0 ⇒ Host 31 : Setup Statistical Generator (Time: 0, Bytes: 0, Count: 0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value: 10000000))

0 ⇒ Host 32 : Setup Statistical Generator (Time: 0, Bytes: 0, Count: 0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value: 10000000))

0 ⇒ Host 33 : Setup Statistical Generator (Time: 0, Bytes: 0, Count: 0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value: 10000000))

0 ⇒ Host 34 : Setup Statistical Generator (Time: 0, Bytes: 0, Count: 0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value: 10000000))
```

Step 4: Establishment of TCP conversation between Hosts in LAN 4 and Hosts in LAN 7.

Set Initial Sequence Numbers for Hosts 41, 42, 43 and 44 and Hosts 71, 72, 73 and 74.

```
0 ⇒ Host 41 : Set Parameters (ISN: 12345678)
0 ⇒ Host 71 : Set Parameters (ISN: 12345678)

0 ⇒ Host 42 : Set Parameters (ISN: 12345678)
0 ⇒ Host 72 : Set Parameters (ISN: 12345678)

0 ⇒ Host 43 : Set Parameters (ISN: 12345678)
0 ⇒ Host 73 : Set Parameters (ISN: 12345678)

0 ⇒ Host 44 : Set Parameters (ISN: 12345678)
0 ⇒ Host 74 : Set Parameters (ISN: 12345678)
```

Request Hosts 41, 42, 43 and 44 to Connect Session to Hosts 71, 72, 73 and 74 (respectively) and Hosts 71, 72, 73 and 74 to Connect Session to Hosts 41, 42, 43 and 44 (respectively).

```
0 ⇒ Host 41 : Connect Session (Addr: Host 71)
0 ⇒ Host 71 : Connect Session (Addr: Host 41)

0 ⇒ Host 42 : Connect Session (Addr: Host 72)
0 ⇒ Host 72 : Connect Session (Addr: Host 42)

0 ⇒ Host 43 : Connect Session (Addr: Host 73)
0 ⇒ Host 73 : Connect Session (Addr: Host 43)

0 ⇒ Host 44 : Connect Session (Addr: Host 74)
0 ⇒ Host 74 : Connect Session (Addr: Host 44)
```

Instruct the Generator on Hosts 41, 42, 43 and 44 to produce a single Constant unit of data.

```
0 ⇒ Host 41 : Setup Statistical Generator (Time: 0, Bytes: 0, Count: 0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value: 10000000))

0 ⇒ Host 42 : Setup Statistical Generator (Time: 0, Bytes: 0, Count: 0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value: 10000000))

0 ⇒ Host 43 : Setup Statistical Generator (Time: 0, Bytes: 0, Count: 0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value: 10000000))
```

```
0 ⇒ Host 44 : Setup Statistical Generator (Time: 0, Bytes: 0, Count:
0, Time (Type: CONSTANT, Value: 1), Space(Type: CONSTANT, Value:
10000000))
```

Step 5: Terminate the simulation at Time 120

Stop.

```
120 ⇒ :
```

The Management script is constructed by translating these pseudo operations using the information provided in Part 1. This is not provided here, as it is cryptic and pointless.

3.4.4.4. Execution: Queue Length Iteration Simulation

The same Parameters and Management script are used as in the Basic Simulation, however for the WAN Queue Length Parameter a BONeS iteration dialog is selected. This dialog is instructed to step through the WAN Queue Length from values 1 to 64 inclusive.

3.4.4.5. Execution: WAN Link Bandwidth Iteration Simulation

The same Parameters and Management script are used as in the Basic Simulation, however for the WAN Link Bandwidth Parameter a BONeS iteration dialog is selected. This dialog is instructed to step through the WAN Link Bandwidth from values 8kbps to 512kbps in 8kbps steps, inclusive.

3.4.5. Expectations (NOT FINISHED)

simulation commences. 12 tcp conversations establish themselves. they fight for position in the network. during this slow start stuff, they lose a lot. we don't really care about them losing a lot. eventually they reach some kind of steady state. but they will still be probing the network. if we loo kat the network, there is the wan link, and the feeders to the wan. the wan link acts as the main resource constraint. the wan feeders do provide space and therefore do allow for more data to be in the network. however it is the central wan at which the bottleneck will occur. the wan link and router can support only 4 segments at any one point in time. 12 conversations must share these 4 segments. consider that if each conversation hada window size of one, there would be 12 packets in the network at any one point in time. these would tend to be fairly evenly spaced out, they can reside in the feeders to or from the wan as well. <xyz> in any case, as we reduce the amount of space available

When the simulation commences execution, we will see 12 TCP conversations attempt to establish themselves across across the path between each LAN and the WAN link. With 12 conversations simultaneously competing for the network resources that are contained within the WAN Router and the WAN Link, congestion should occur fairly quickly. Therefore, the initial expectation is that there will be quite a high level of congestion. This transient isn't of particular concern, as it is the steady state behaviour that is of interest here.

Eventually, the conversations will reach some form of equilibrium. In the basic simulation the WAN Router has a Queue Length of 4, and the WAN Link has a bandwidth of 16kbps with a propagation delay of 100ms. Although each conversation passes through an connection before it reaches the WAN Router, this connection will not act as the bottleneck, though it will introduce delay, and contribute to the total round trip time of the connection. The bandwidth delay of the WAN is 200 bytes, which is less than one segment. Therefore, all 12 conversations must share a total of 4 segments through the WAN.

- we can predict that the rtt will be of a certain value

- can predict that the congestion windows and slow start thresholds should be value x.

- the following diagram illustrates the epected window behaviour

- note that the round trip time is bounded

- with the wan acting as the bottleneck, lets look at what happens in the wan feeders

- they contribute a certain amount to the round trip time, but they will never become

congested

- the wan router queue information should be fully utilised

- we have to consider that packets will come through interspaced, and interleaved

- this means that for the certain rtt, we send packets of this size

- but what happens when we play around with the rtt through alteration of the wan queue

- if we decrease queueing so that there can be only one packet, then all the conncetions must share one packet space over

- of more interest is the

- this is the case for a single conversation though

For the moment, we can ignore the queueing space. The 12 conversations start with a window of 2, and with Transport Layer, Network Layer and Datalink Layer overhead of 32 bytes in total, each packet transported on the Link is of size 34 bytes. 12 packets of 34 bytes total 408 bytes, so all packets will fit into the network. It is when the windows increases from 32 bytes to 64 bytes that congestion occurs (giving rise to 12 packets at 64+32 bytes each, exceeding the total of 800).

With a queue length of 2, not all conversations will be subject to congestion at this window size, but generally the congestion windows should stay at these low values. congestion will tend to occur frequently, consider that if the maximum usable window is about 34 bytes, then windows will potentially average around 15-20 bytes, and increase by 15-20 bytes before suffering congestion. an increase of 20 bytes would take 20 round trip times, where each rtt is approximately 800ms (it is dominated bt the propagation delay in this case), i.e. 16 seconds. We can predict that the congestion window nad slow start threshold exhibit this behaviour:

<x>

As an approximation, 25 bytes are sent every 800ms for 16 seconds, upon which the loss of 30 bytes occurs. So, for a transfer of 500 bytes, 30 bytes are lost, about 6%. The graphs are shown in the following diagram:

<x>

These results are more interesting when considered in light of iterations on WAN Link Propagation Delay. With a lower propagation delay, less bytes are capable of fitting into the network, therefore the average congestion window and slow start thresholds are lower, and as a consequence our loss rates are higher. To illustrate this, consider the propagation delay that is now 300ms. The delay bandwidth is (16kbps/8 * 300ms) = 600 bytes, so congestion will occur when packets reach a size of 50 bytes. Removing the 32 byte overhead due to layering, the maximum TCP payload is 18 bytes. Therefore, the windows will potentially average around 9 bytes, and increase by 9 bytes berfore suffering congestion. an increase of 9 bytes would take 9 round trip times, where each rtt is now approprlately 600ms, i.e. 5 seconds.

As an approximation, 9 bytes are sent every 600ms for 5 seconds, after which a loss of 18 bytes occurs. So, for a transfer of 75 bytes, 18 are lost, about 24%. This illustrates the potential problem with the overloaded network. As we decrease the propagation delay, a point will be reached where continual retransmissions occur, corresponding to the case where congestion potentially occurs on every single transmission. We can attempt to predict this threshold and the effects.

consider that with 12 conversations, and the transfer of 1 byte per RTT where each packet has a 32 byte overhead, the delay bandwidth product must be capable of hodling a total of 396 bytes. If it cannot ...

Upon iteration of the queue length, we tend to expect better results at the expense of bandwidth.

3.4.6. Execution of Simulation

The simulation was not executed due to the problems surrounding the unavailability of the BONeS software.

3.4.7. Analysis of Results

No results were gathered from the simulation due to the problems surrounding the unavailability of the BONeS software.

3.4.8. Conclusions (NOT FINISHED)

our expectations do show that there is cause for concern. although we used a low number of conversations, the basic principle is that the network is not capable of supporting data from each window every round trip time. as the network becomes less capable, the amount of loss increases considerably, as measured by the level of retransmissions. beyond a given threshold, the losses are substantial and the network is virtually dominate by retransmissions.

3.5. Fluctuating traffic on TCP conversations through bottleneck long -haul WAN Link

3.5.1. Problem and Objectives

This scenario concerns itself with the third major issue identified in relation to the Transmission Control Protocol's (TCP) congestion control mechanisms as they apply in Wide-Area Network (WAN) environments.

The concern is based upon the shift in traffic characteristics that are being seen in WANs. In particular, WAN traffic previously consisted mainly of two types of TCP conversations: medium to long term sustained transfers, such as bulk data FTP, and bursty poisson conversations, such as interactive Telnet sessions.

More recently, the emergence and predominance of the World-Wide Web (WWW) has shifted WAN traffic profiles due to the particular characteristics of the WWW. The WWW primarily uses a session-based protocol called the Hyper-Text Transfer Protocol (HTTP). It uses a TCP conversation to make short requests and responses.

The TCP congestion control mechanisms use closed loop feedback received over a number of Round Trip Times (RTTs). It uses this information to determine network operating conditions which form the basis of its control algorithm. The short lived nature of the WWW's HTTP based conversations is such that the underlying TCP conversation does not exist for a time long enough to receive sufficient information from the network, and therefore does not adequately accommodate the network.

Therefore, *the objective in these scenarios is to examine the effects of HTTP based conversations to determine whether or not there is concern about the lack of congestion control in such conversations.*

3.5.2. Approach

The approach consists of identifying the model, simulation and observations that are required to obtain the objectives.

Model

The same model parameters are used as in the previous simulation.

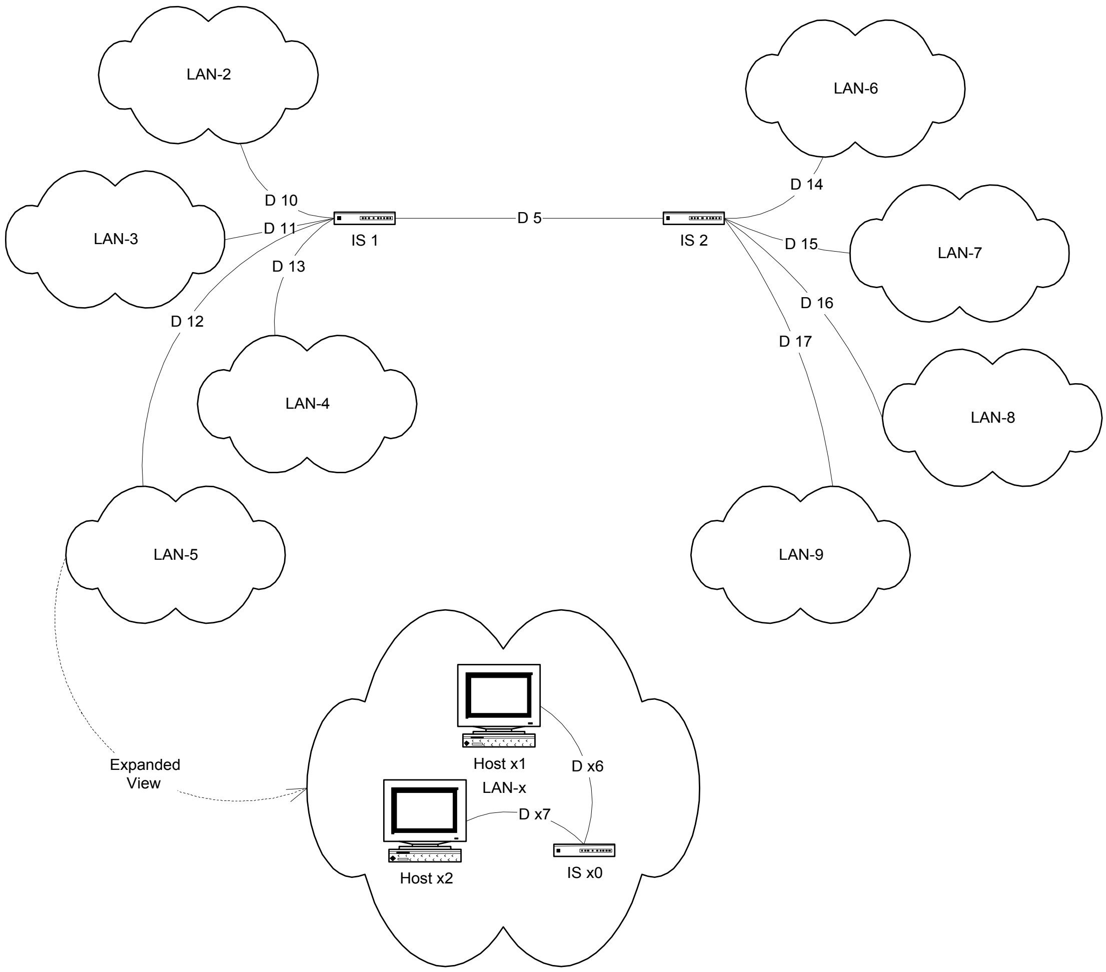


Figure 2-3.10. Simulation Model: Multiple TCP Conversation F-WAN

Basic Simulation

The basic simulation consists of many conversations of a short nature continually occurring between the Hosts in LAN 2, LAN 3, LAN 4, LAN 5 and the Hosts in LAN 6, LAN 7, LAN 8, LAN 9 respectively. In addition, there are two conversations that exist between LAN 2 and LAN 5. These conversations carry one way traffic (a transfer of a large unit of data) in the forward direction, and the return direction consists only of acknowledgements. This runs for 120 seconds.

This phase represents the case of two TCP conversations with long lifetimes that suffer the effects of many short-lived HTTP type conversations. In the investigation, interest is in the losses and throughputs related to the TCP transmitters, especially the bursty conversations as their losses will be large in comparison to their data transferred. The Queue Length for the WAN Routers are captured, to ensure that it shows that congestion is occurring.

The bursty traffic consists of conversations that are started at random time intervals, and consist of a short transfer (128 bytes) in one direction, a delay, and a return of a larger transfer (2048 bytes) in the return direction. This is an attempt to model HTTP traffic, as it consists of a simple request, and the return of a "page" of information.

The simulation is stopped after 120 seconds.

Variations

There are three variations of concern.

The first involves the effects of altered Queue Lengths in the WAN Routers. It is expected that the bursty nature of the conversations is better accommodated with larger queues; therefore simulations are run with Queue Lengths between 1 and 64 (packets).

The second involves replacing the HTTP traffic with Telnet traffic, as it is desired to see whether the effects of either have similar impact upon the network. This occurs by having the random conversations employ a Telnet profile, rather than use the request and response mechanism. The Telnet traffic is generated in both directions.

The third involves repeating the Basic Simulation and the first two variations with different Queue Disciplines: Random Drop and Random Early Detection.

Observations

The observations are gained from two sources. The TCP transmitters, all of them, provide information about their congestion control characteristics and more general items such as retransmission levels. From these, we can gather the extent to which the transmitter is receiving feedback information from the network.

The WAN Router provides an indication of Queue Lengths, which illustrate congestion. In particular, the share given to the two sustained transfers is of interest.

3.5.3. BONeS Simulation Design

3.5.3.1. Topology and Parameter Values

This section was not completed.

3.5.3.2. Runtime Management Script

This section was not completed.

3.5.3.3. Probes and Post Processing

This section was not completed.

3.5.3.4. Execution

This section was not completed.

3.5.4. Expectations

This section was not completed.

3.5.5. Execution of Simulation

This section was not completed.

3.5.6. Analysis of Results

This section was not completed.

3.5.7. Conclusions

This section was not completed.

CONCLUSIONS

By looking at our objectives, and the work carried out in this thesis, we can make some interesting conclusions. Unfortunately, the problems that manifested themselves prevent these conclusions from being more substantial and justifiable in nature.

The BONeS environment was used to carry out the entire process of modelling, simulating and result processing. The experience with it has been extremely positive, as it is extremely easy to use, flexible and complete in ability. Its encapsulation of the entire lifecycle, and automation of simulation and result processing reduce the run-analyse-debug cycle. Intrinsic support for iteration, and complex manipulation of data sets in post process allow for fast and comprehensive examination of variations.

There were some minor problems with BONeS, but these tended to be trivial in nature. The biggest complaint, by far, is the attitude and policy of Comdisco (the producers of BONeS), which resulted in the fiasco that prevented BONeS from being usable for significant part of the thesis.

It can be seen in this report that significant effort was put into the construction and documentation of the BONeS environment. This more than meets the objectives of providing a flexible, presentable and re-usable environment. The expansion occurred largely as a result of the inability to carry out other work in thesis, in the attempt to salvage something of use.

The main concern with modelling and simulation was to examine the effects of changing WAN environments in relation to TCP congestion control. Through the construction of simulations and the examination of expected results, it can be concluded that these new environments do impact upon the performance of TCP congestion control. These conclusions are based upon the expectations of the simulations, as the unavailability of BONeS did not allow any actual simulations to be carried out.

The first two simulations were less important in nature, as they examined behaviour that is already understood. However, they did allow us to see the operation of TCP congestion control, and develop the insights in our own particular manner. The objectives relating to verification and validation of or models based upon the simulation results were not reached, due to BONeS being unavailable, however the objective to explain the nature of TCP congestion control was reached. Other work has not attempted to provide the type of qualitative analysis given here.

The third simulation looked at the effects of increasing WAN size and complexity, in an attempt to determine the effects of such on TCP congestion control. Although no simulation was performed, the expectations lead to the belief that these networks do cause performance problems, most notable due to retransmissions generated through out of order delivery and variant round trip time estimations.

The fourth simulation was concerned with the increasing traffic levels placed upon individual WAN connections, and a potential problem that can result due to the low granularity of the TCP congestion control window. Basically, the network can only support a transmission rate from each transmitter that is lower than the minimum that can be provided through TCP congestion control. The expected result is as this supportable rate decreases, retransmission reach unacceptable levels, causing significant performance losses. Increasing queuing is expected to alleviate the situation, as a short-term measure, but a longer-term measure must consider a TCP congestion control strategy that can transmit at lower rates.

The fifth simulation probes the effects of new traffic classes becoming dominant in the WAN environment, specifically the case of the HTTP, which uses TCP to carry out short bursty conversations (in a transaction nature). These conversations do not exist for long enough to adequately assess and interoperate with the network in terms of congestion conditions. When coupled with highly utilised WAN connections, the expected situation is that considerable losses occur. It is also expected that these results are particular to the HTTP, and not apparent with seemingly similar bursty traffic, such as Telnet.

In conclusion, this work has not met all of its objectives, due mostly to unforseen circumstances. However, although it wasn't possible to run any simulations, issues have been identified and examined and a framework has been presented allowing for sufficient assessment of whether the basis for the issues are legitimate. Sufficient work has been done to allow others to take up the challenge of pressing ahead.

As a final note, it is interesting to consider the nature of the problem that occurred with the BONeS software. The lesson taught is that if you are carrying out work using some item of equipment, be it software or otherwise, always ensure that if the item fails, then there is a means to continue without the loss of significant effort. Principally, this indicates that the assumptions that seen the most stable, may not be so, and tends to summarise the nature of the issues in this work.

FUTURE DIRECTIONS

The conclusions reached, and investigation that occurred during this work has led to some ideas about future directions that could be taken based upon the investigations here, and the topic in a more general sense.

In the most obvious case, the incomplete investigations could be completed. This would require the execution and analysis of the simulations, either using the BONeS environment set up here (which is preferable, given the work that has been put into it) or in some other manner. The results could then be measured up against the expectations, and more concrete analysis could be performed.

Thesis 1 mentioned that a potential goal for this work was to perform a comparison of TCP congestion control using the various algorithms that have been developed. With the proliferation of many new congestion control algorithms in the last few years, a comprehensive treatment is even more pressing. Such examination should try to provide both a theoretical and practical analysis, and look deeper at the underlying concepts of these algorithms, in terms of control theory and so on. If all models and simulations in such an examination were equivalent, then it would provide the first comprehensive examination.

The potential solutions to problems identified in this work should be examined to determine their success. The two most significant solutions identified in this work are the alteration of TCP to use a "super slow start window", in order to alleviate the "window-granularity problem"; and the use of TCP with Transaction Extensions as a protocol for HTTP operation. The solutions can be assessed through both simulations and actual implementation.

The specific characteristics of WANs also need to be looked at further. This would require more simulations with realistic scenarios, better models of WAN traffic, and investigations with the new TCP transaction protocol (to assess its interoperability in existing environments, for the purpose of gauging how successful its deployment could be). It has been noted [ref] that existing models and simulations have failed to adequately represent the complex networks and characteristics visible in the current environment.

Undoubtedly the best direction to take would involve attempting to look at the results obtained here, both as expectations and through the execution of the simulations, and attempt to verify these would observations from operating WANs. Although there are a lot of difficulties in obtaining statistics from operating WANs (politically and technically), such a task can be achieved, would be a challenge and would validate issues raised in this work.

At this point in time, the TCP has been in existence for more than 10 years, and Van Jacobson's seminal congestion control work is over 6 years old. The last few years have seen the most rapid changes in WAN architecture and use, with more changes expected to continue. More than ever, there are many pressing issues to look at, as these changes challenge strategies developed for a different world.

REFERENCES

Braden, R. (1989). Requirements for Internet hosts -- Communications layers, *Network Working Group Request for Comments RFC 1122*, Internet Engineering Task Force.

Braden, R. (1994). T/TCP - TCP Extensions for Transactions, Functional Specification, *Network Working Group Request for Comments RFC 1644*, Internet Engineering Task Force.

Brakmo, L. S., O'Malley, S. W. and Peterson, L. L. (1994). TCP Vegas: New techniques for congestion detection and avoidance, *Technical report*, Department of Computer Science, University of Arizona, Tucson, AZ.

Brakmo, L. S. and Peterson, L. L. (1995). Performance Problems in BSD4.4 TCP, *Technical report*, Department of Computer Science, University of Arizona, Tucson, AZ.

Cáceres, R., Danzig, P. B., Jamin, S. and Mitzel, D. J. (1991). Characteristics of Wide-Area TCP/IP Conversations, *SIGCOMM Symposium on Communications Architectures and Protocols*, ACM, Zürich, Switzerland, pp. 101--112. also in *Computer Communication Review* 21 (4), Sep. (1991).

Comer, D. E. (1991). *Internetworking with TCP/IP Vol II: Design, Implementation, and Internals*, Prentice Hall, Englewood Cliffs, NJ.

Berkeley Software Distribution. (1994). The 4.4BSD-Lite release. *ftp://ftp.cdrom.com/pub/bsd-sources/4.4BSD-Lite.tar.gz*

Chiu, D.-M. and Jain, R. (1989). Analysis of the increase and decrease algorithms for congestion avoidance in computer networks, *Computer Networks and ISDN Systems*, Vol. 17, pp. 1--14.

Comdisco Systems Inc. (1993). *BONeS DESIGNER Users Guide V2.1.*

Danzig, P. B., Lin, Z. and Limin, Y. (1995). An Evaluation of TCP Vegas by Live Emulation., *Technical report*, Computer Science Department, University of Southern California, Los Angeles, CA.

end2end-interest, (1988)+. End-to-End interest mailing list, *ftp://ftp.isi.edu/end2end.*

Floyd, S. (1991a). Connections with multiple congested gateways in packet-switched networks. Part 1: One-way traffic, *Technical report*, Lawrence Berkeley Laboratory.

Floyd, S. (1991b). Connections with multiple congested gateways in packet-switched networks. Part 2: Two-way traffic, *Technical report*, Lawrence Berkeley Laboratory.

Floyd, S. (1994). TCP and Explicit Congestion Notification, *Technical report*, Lawrence Berkeley Laboratory.

Floyd, S. (1994). Simulator Tests, *Technical report*, Lawrence Berkeley Laboratory.

Floyd, S. (1995). TCP and Successive Fast Retransmits, *Technical report*, Lawrence Berkeley Laboratory.

Floyd, S. and Jacobson, V. (1992). On traffic phase effects in packet-switched gateways, *Internetworking: Research and Experience* 3 (3): 115--156.

Floyd, S. and Jacobson, V. (1993). Random early detection gateways for congestion avoidance, *IEEE/ACM Transactions on Networking* 1 (4): 397--413.

Floyd, S. and Paxson, V. (1994). Wide-Area Traffic: The Failure of Poisson Modeling, *Technical report,* Lawrence Berkeley Laboratory.

Huynh, L., Chang, R.-F. and Chou, W. (1994). Performance comparison between TCP slow-start and a new adaptive rate-based congestion avoidance scheme, *Proc. 2nd Internationl Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, IEEE, pp. 300--307.

International Organisation for Standardisation. (1984). Basic Reference Model for Open Systems Interconnection, ISO 7478.

Jacobson, V. (1988). Congestion avoidance and control, *ACM Computer Communication Review* 18 (4): 314--329. Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, (1988).

Jacobson, V. (1990). Modified TCP Congestion Avoidance Algorithm, end2end-interest mailing list.

Jain, R. (1990). Congestion Control in Computer Networks: Issues and Trends, *IEEE Network Magazine*.

Jain, R. and Ramakrishnan, K. K. (1988). Congestion avoidance in computer networks with a connectionless network layer: Concepts, goals and methodology, *Proc. Computer Networking Symposium*, IEEE, pp. 134--143.

Mogul, J. C. (1992). Observing TCP Dynamics in Real Networks, *SIGCOMM Symposium on Communications Architectures and Protocols*, ACM, Baltimore, Maryland, pp. 305--317. *Computer Communication Review*, Volume 22, Number 4.

Nagle, J. (1984). Congestion control in IP/TCP internetworks, *Request for Comments RFC 896*, Internet Engineering Task Force.

Nagle, J. (1987). On packet switches with infinite storage, *IEEE Transactions on Communications* COM-35 (4): 435--438.

Paxson, V. (1991). Measurements and Models of Wide Area TCP Conversations, *Technical report*, Department of Computer Systems Engineering, University of California, Berkeley, CA.

Paxson, V. (1993a). Growth Trends in Wide-Area TCP Connections, *Technical report*, Lawrence Berkeley Laboratory.

Paxson, V. (1993b). Empirically-Derived Analytic Models of Wide-Area TCP Connections: Extended Report, *Technical report,* Lawrence Berkeley Laboratory and EECS Division, University of California, Berkeley, CA.

Postel, J. (1981a). Internet Protocol, *Network Working Group Request for Comments RFC 791*, Information Sciences Institute, University of Southern California.

Postel, J. (1981b). Internet Control Message Protocol, *Network Working Group Request for Comments RFC 792*, Information Sciences Institute, University of Southern California.

Postel, J. (1981c). Transmission Control Protocol, *Network Working Group Request for Comments RFC 793*, Information Sciences Institute, University of Southern California.

Postel, J. B. (1982). Simple Mail Transfer Protocol, *Network Working Group Request for Comments RFC 821*, Internet Engineering Task Force.

Postel, J. B., Reynolds, J. K. (1985) File Transfer Protocol (FTP), *Network Working Group Request for Comments RFC 959*, Internet Engineering Task Force.

School of EE UTS. (1994). *UTS Electrical Engineering Thesis Subjects*, A/94 edn.

Shanmugan, K. S., LaRue, W. W., Klomp, E., McKinley, M., Minden, G. J. and Frost, V. S. (1988). Block-Oriented Network Simulator (BONeS), *Conference Record, GLOBECOM 88*, Vol. 3, IEEE, pp. 1679--1684.

Stallings, W. (1993). *Networking Standards : A Guide to OSI, ISDN, LAN and MAN Standards*, Addison-Wesley.

Stevens, W. R. (1994). *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley.

Stevens, W. R. and Wright G. R. (1994). *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley.

Tipper, D., Hammond, J. L., Sharma, S., Khetan, A., Blakrishnan, K. and Menon, S. (1994). An Analysis of the Congestion Effects of Link Failures in Wide Area Networks, *IEEE Journal on Selected Areas in Communications* 12 (1): 179--192.

Wang, Z. (1992). *Routing and Congestion Control in Datagram Networks*, PhD thesis, University College London.

Wang, Z. (??). A dual-window model for flow and congestion control, *Technical report*, Dept Computer Science, University College London.

Wang, Z. and Crowcroft, J. (1991). A new congestion control scheme: Slow Start and Search (Tri-S), *Computer Communication Review* 21 (1): 32--43.

Wang, Z. and Crowcroft, J. (1992). Eliminating periodic packet losses in the 4.3-Tahoe BSD TCP congestion control algorithm, *ACM Computer Communication Review* 22 (2): 9--16.

Zhang, L., Shenker, S. and Clark, D. D. (1991). Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic, *Sigcomm '91 Conference: Communications Architectures and Protocols*, ACM, Zürich, Switzerland, pp. 133--147.

APPENDIX 1. DETAILED BONES DESIGN

1. Data Structures

1.1. Messages

Messages are used for communication between Modules and are constructed as COMPOSITE types. They are structured so that there is a common Message Primitive from which all Messages are derived, which enhances the type checking capability in the environment.

The Message Primitive defines two fields: a Length and a Creation Time. The Length is used to compute a delay for the Message, and the Creation Time to monitor the lifetime of a Message (e.g. a TCP packet). The Length effectively models the amount of Data that is present in the Message.

The top level consists of a “Message Primitive”. The next level defines primitives for each particular Module that has associated Messages: Datalink Layer, Network Layer, Transport Layer and Management. For example, the “Datalink Message Primitive” for the Datalink Layer. Within each particular Module, i.e. the next level, Messages are further partitioned according to function type, e.g. “Connect Primitive” and “Disconnect Primitive”. At the final level is the particular Message qualified for a particular function, e.g. “Request” and “Indication”.

“Data” Messages always encapsulate other Messages; whereas Messages for operational purposes (e.g. “Connect” and “Disconnect”) generally encapsulate Information Elements.

1.2. Information Elements

Information Elements are used to convey specific units of data between Modules, and they are generally encapsulated within Messages. The Information Element is constructed as a COMPOSITE type. They are structured similar to Messages in terms of layout.

The top level consists of an “Information Element Primitive”. The next level defines primitives for each particular Module that has associated Information Elements: Datalink Layer, Network Layer, Transport Layer, Network-Adaption Layer, Transport-Adaption Layer, Routing-Module, Generator and Management. Within each particular Module, i.e. the next level, the Information Elements are defined in whichever manner is appropriate for them.

1.3. Miscellaneous

For simplicity, a Boolean type is constructed as a SET of “True” and “False”, it is used as an alternative to a less tightly constrained INTEGER.

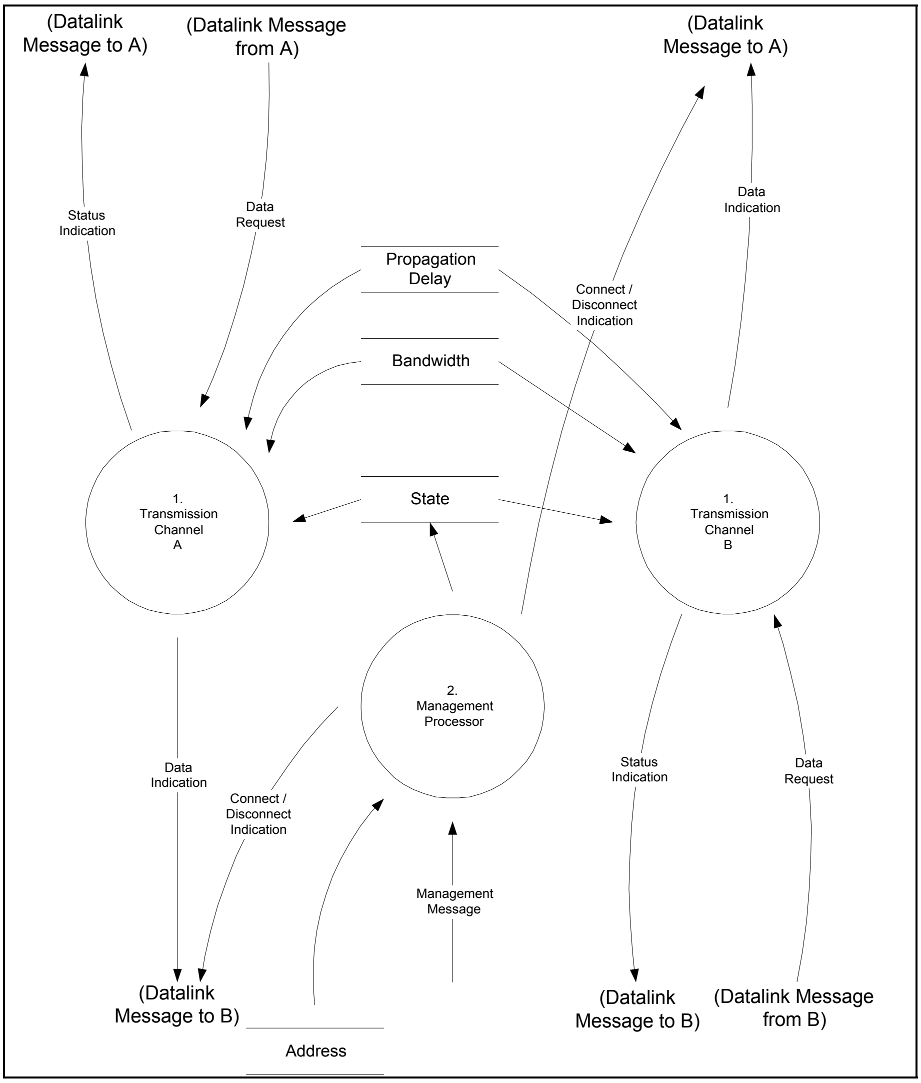
“Generate Statistical Parameter” requires Data Structures for its operation; this involves a base COMPOSITE type for a generalised “Statistical Parameter” and specific Statistical Parameters (e.g. “Constant” and “Exponential”) are constructed as derivations from this base.

2. Primary Modules

2.1. Datalink Layer

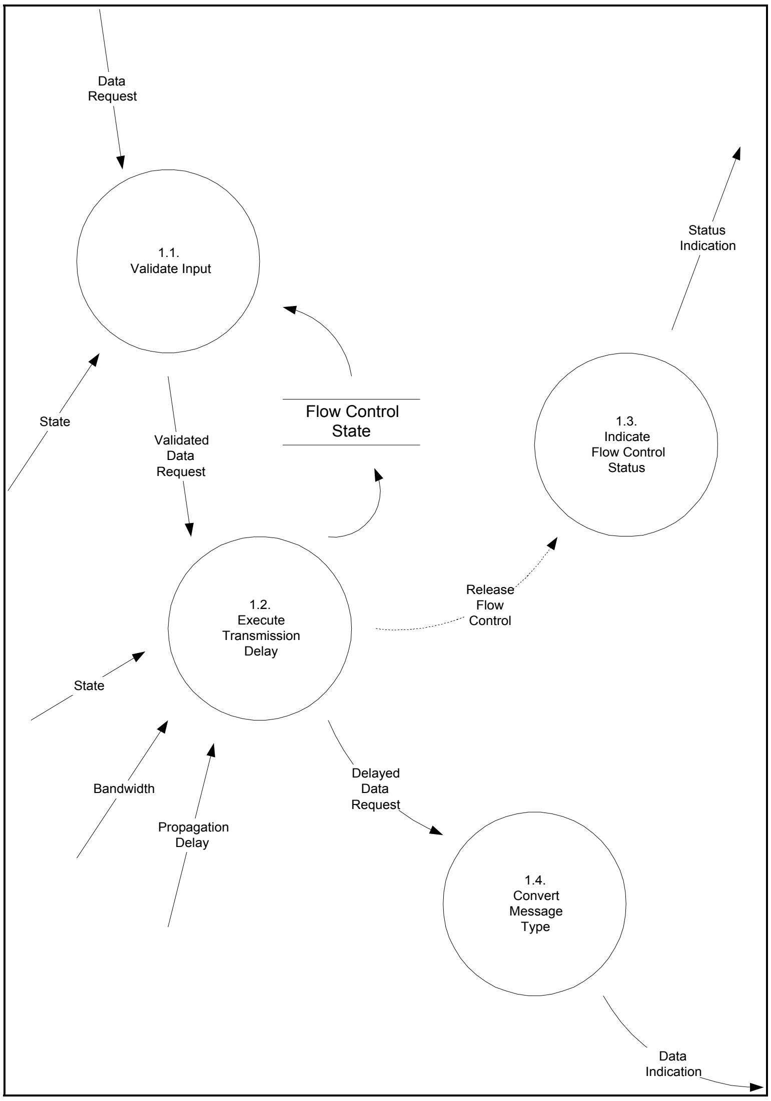
DFD 0: Top

The Top Level DFD shows the Transmission Channels and Management Processor with specific delineation of data relationships between them, and the upper layer entities.



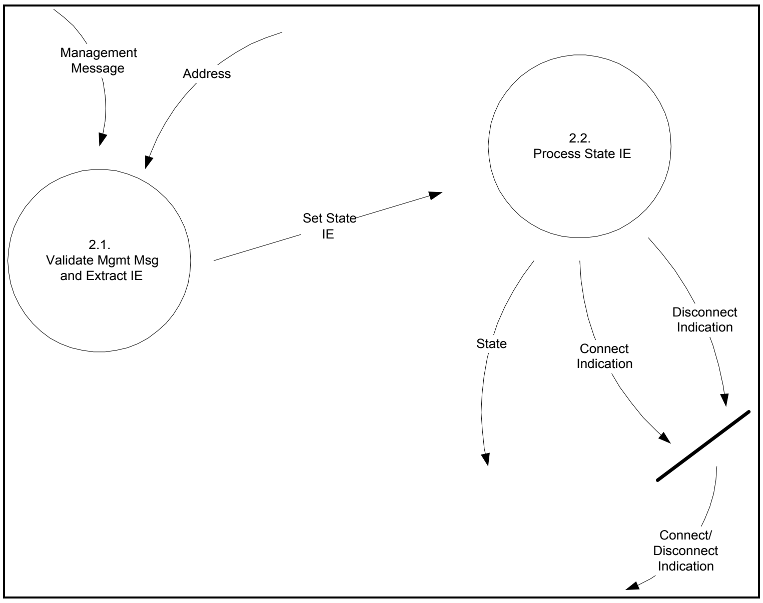
DFD 1: Transmission Channel

The *Transmission Channel* has the task to first *Validate Input*, thence to *Execute Transmission Delay*--involving *Indicate Flow Control* notification--after which a *Convert Message Type* occurs. Although it is possible to indicate both cases of the Flow Control being *Released* or *Asserted*; it is presumed that as soon as a message is sent, it is *Asserted* until an explicit *Release* occurs via this module.



DFD 2: Management Processor

The *Management Processor* has a straight forward partitioning. Firstly, the message is validated in *Validate Mgmt Message and Extract IE* to ensure that it has the correct destination address and content, after which the content is processed according its type. The only specific content processed at this point in time is the *Datalink Set State IE* in *Process State IE* which results in a new *State*.



PSPEC 1.1: Validate Input

The message is validated before it is processed by using *Validate Input*. This validation is only concerned with the *State* of the Datalink and the *Flow Control State* of the Channel. If either of these tests fail, then the message is discarded.

```
Inputs:
    State: Boolean
    Flow_Control_State: Boolean
    Data_Req: Datalink Data Request Message
Outputs:
    Validated_Data_Req: Datalink Data Request Message
Operation:
    1. IF State = True THEN
        1. IF Flow_Control_State = True THEN
            1. Validated_Data_Req := Data_Req
    2. STOP
```

PSPEC 1.2: Execute Transmission Delay

The message is delayed in time due to the modelling of transfer across a media, by using this *Execute Transmission Delay*. This involves first a delay due to the transmission of the message, a result of the *Bandwidth* of the Channel and the message *Length*, and then a delay due to *the Propagation Delay* of the Channel.

```
Inputs:
    Validated_Data_Req: Datalink Data Request Message
    State: Boolean
    Bandwidth: Integer
    Propagation_Delay: Real
Outputs:
    Delayed_Data_Req: Datalink Data Request Message
    Flow_Control_State: Boolean
    Release_Flow_Control: SIGNAL
Operation:
    1. Flow_Control_State := False
    2. SLEEP ( Length (Validated_Data_Req) / Bandwidth )
    3. IF State = True THEN
        1. Flow_Control_State := True
        2. SIGNAL ( Release_Flow_Control )
        3. SLEEP ( Propagation_Delay )
        4. IF State = True THEN
            1. Delayed_Data_Req := Validated_Data_Req
    4. STOP
```

PSPEC 1.3: Indicate Flow Control Status

When activated, *Indicate Flow Control Status* will generate a status message indicating that flow control is released.

```
Inputs:
    Release_Flow_Control: SIGNAL
Outputs:
    Status_Ind: Datalink Status Indication Message
Processing:
    1. DECLARE Flow_IE: Datalink Flow Control IE
    2. Flow_IE := ConstructIE_DL_Flow_Control (Released)
    3. Status_Ind := ConstructMsg_DL_Status_Ind (Flow_IE)
    4. STOP
```

PSPEC 1.4: Convert Message Type

In *Convert Message Type*, the message must be converted from a request to an indication, this is done using a pre-constructed data accessor.

```
Inputs:
    Delayed_Data_Req: Datalink Data Request Message
Outputs:
    Data_Ind: Datalink Data Indication Message
Operation:
    1. Data_Ind := ConvertMsg_DL_Data_Req_To_Ind (Delayed_Data_Req)
    2. STOP
```

PSPEC 2.1: Validate Mgmt Message and Extract IE

The Management message is inspected to ensure that is destined for this module, and that the content is valid via *Validate Mgmt Message and Extract IE*. The appropriate output is generated depending on the type of content.

```
Inputs:
    Mgmt_Msg: Management Set Indication Message
    Address: Integer
Outputs:
    State_IE: Datalink State IE
Processing:
    1. DECLARE Unknown_IE: IE
    2. IF Address (Mgmt_Msg) = Address THEN
        1. Unknown_IE := ExtractMsg_Mgmt_Set_Ind (Mgmt_Message)
        2. IF Type (Unknown_IE) = Type (State_IE)
            1. State_IE := Unknown_IE
    3. STOP
```

PSPEC 2.2: Process State IE

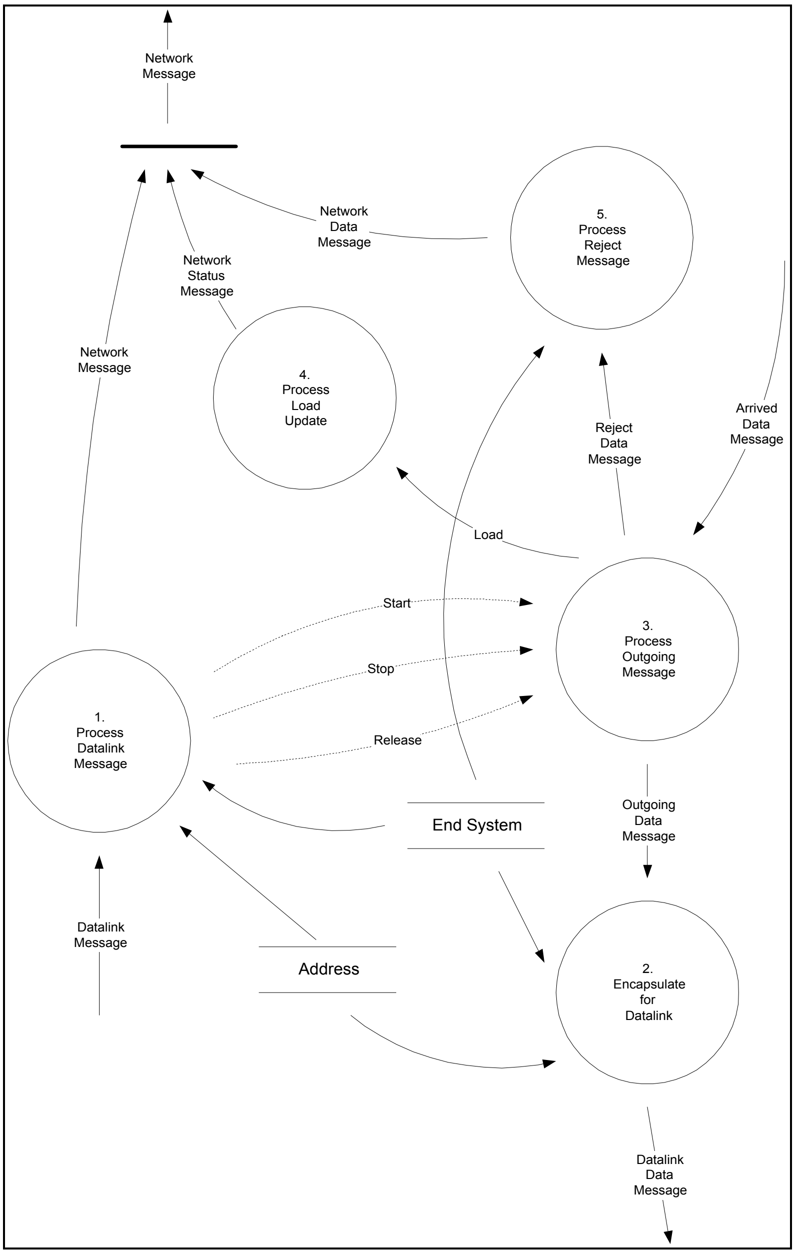
The content of the *State IE* is processed in *Process State IE* and used to update internal state along with generating an indication to upper layers.

```
Inputs:
    State_IE: Datalink State IE
Outputs:
    Connect_Ind: Datalink Connect Indication Message
    Disconnect_Ind: Datalink Disconnect Indication Message
Processing:
    1. DECLARE New_State: Boolean
    2. New_State := ExtractIE_DL_State (State_IE)
    3. If New_State != State THEN
        1. State := New_State
        2. IF State = True THEN
            1. Connect_Ind := ConstructMsg_DL_Connect_Ind ()
        3. IF State = False THEN
            1. Disconnect_Ind := ConstructMsg_DL_Disconnect_Ind ()
    4. STOP
```

2.2. Network Layer

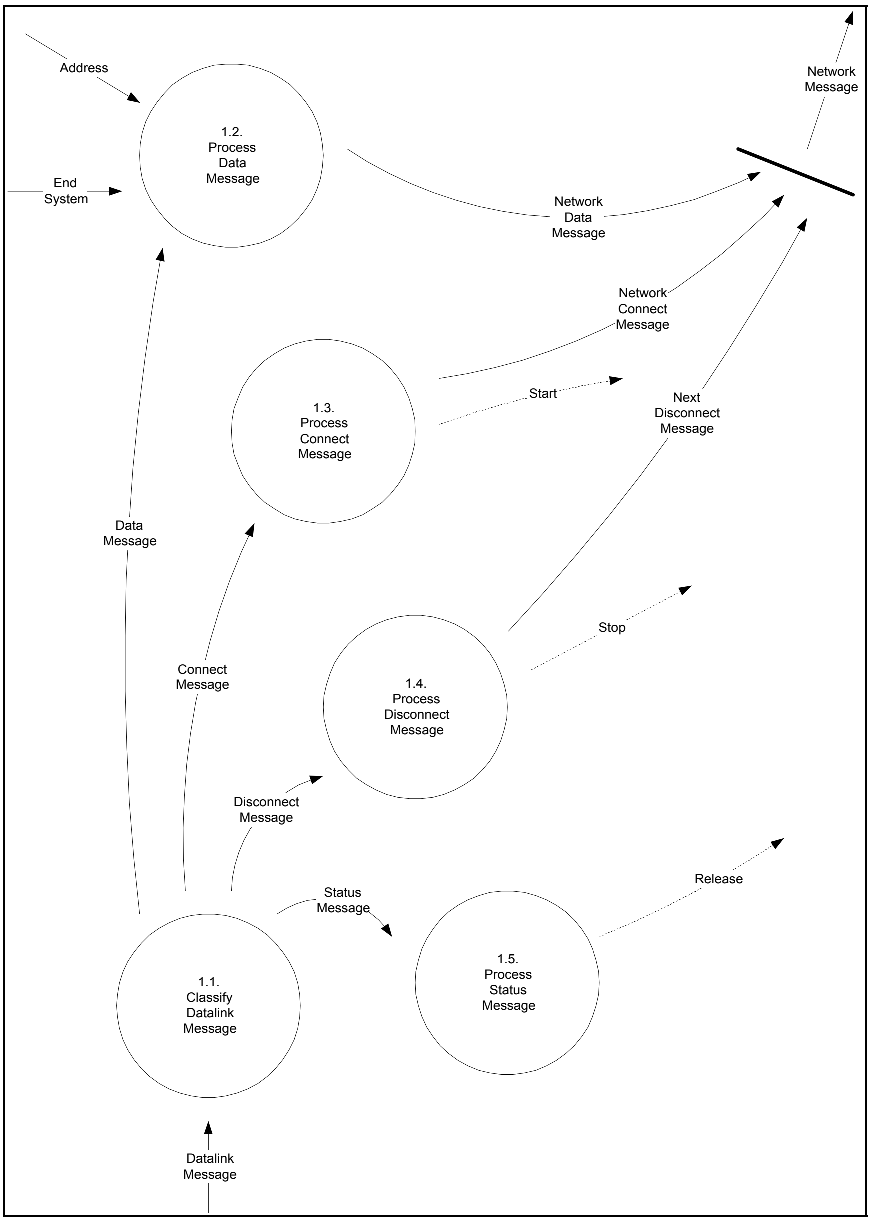
DFD 0: Top

The Top Level DFD shows the major top level processing blocks. The inbound processing is encapsulated within *Process Datalink Message* and the significant outbound processing within *Process Outgoing Message*. *Process Load Update*, *Process Reject Message* and *Encapsulate for Datalink* are outbound processing functions but are not placed within *the Process Outbound Message* because changed Queue policies should only affect one block, and the processing that is static for all types of Queue policies should remain outside of this block, otherwise unnecessary duplication occurs. The *End System* flag alters behaviour *in Process Reject Message* and *Encapsulate for Datalink*: it is set when the Layer is being used in an *End System* situation.



DFD 1: Process Datalink Message

The *Process Datalink Message* is responsible for interpreting and acting upon messages arriving from the Datalink Layer. A *Datalink Connect Indication Message* or *Datalink Disconnect Indication Message* is used to generate a respective *Network Connect Indication Message* or *Network Disconnect Indication Message* for the Upper Layer. Also, respective *Start* or *Stop* triggers are generated for the Outbound Processing. *A Datalink Status Indication Message* contains a Flow Control Release indication that is used to trigger a *Release* for Outbound Processing. A *Datalink Data Indication Message* has the internal *Network Data Indication Message* removed; which is then propagated upwards (if certain conditions prevail).



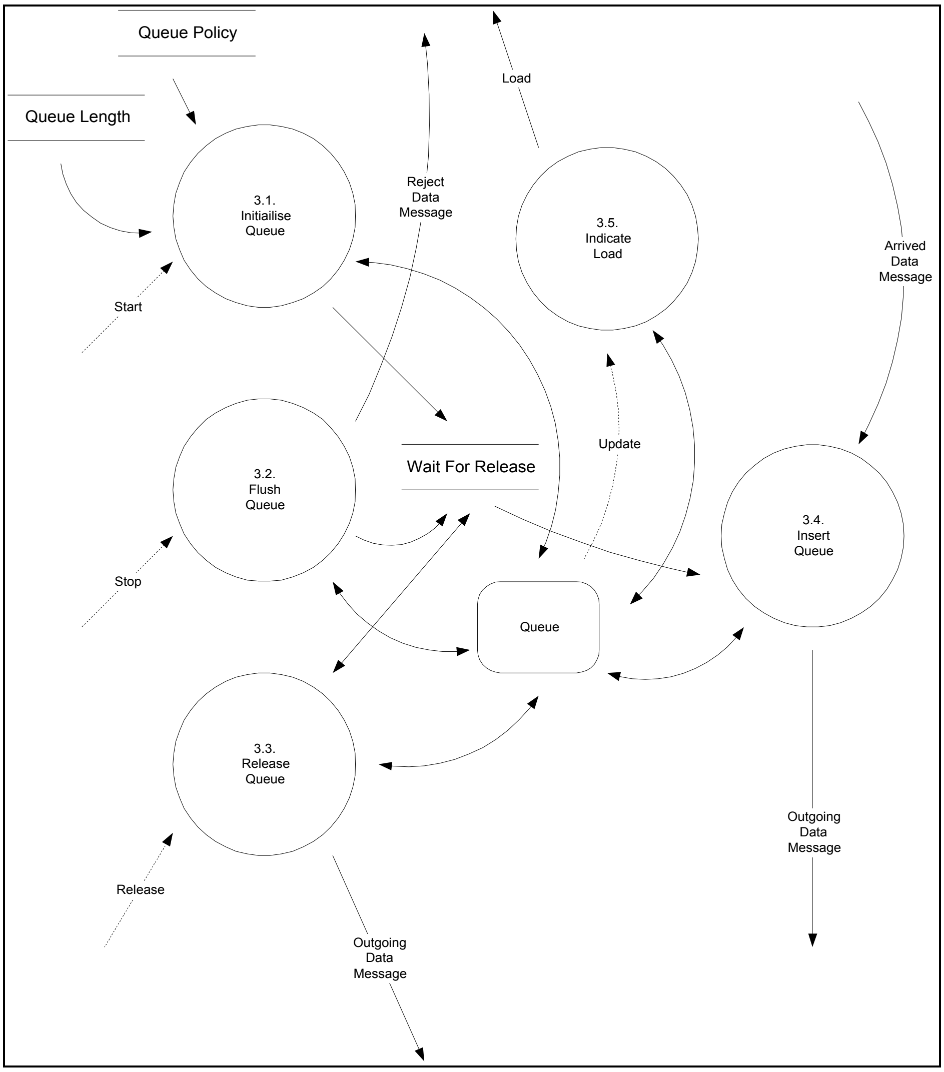
PSPEC 2: Encapsulate for Datalink

Encapsulate for Datalink has the responsibility of taking an *Outgoing Data Message* and placing it within a *Datalink Data Message*, also ensuring that if this Network Layer is in an *End System*, then the source address in the *Outgoing Data Message* must be set correctly.

```
Inputs:
    Outgoing_Data_Msg: Network Data Request Message
    End_System: Boolean
    Address: Integer
Outputs:
    Datalink_Data_Msg: Datalink Data Request Message
Operation:
    1. IF End_System = True THEN
        1. InsertMsg_N_Data_Req (Outgoing_Data_Msg,
                   SOURCEADDRESS, Address)
    2. Datalink_Data_Msg :=
                   ConstructMsg_DL_Data_Req (Outgoing_Data_Msg)
    3. STOP
```

DFD 3: Process Outgoing Message

Process Outgoing Message is the core functionality of the Network Layer. When initialised by *Start*, *Initialise Queue* will use the defined *Queue Policy* and *Queue Length* to set up a *Queue* instance. Any subsequent *Arrived Data Messages* are placed into the *Queue* by *Insert Queue*, and dropped if they cannot fit. A *Release* triggers *Release Queue* that will either allow the next *Outgoing Data Message* to be sent, or indicate that there is no need to *Wait for Release* on the next arrived one. When *Stop* occurs, all items in the *Queue* are output by *Flush Queue* as *Reject Data Messages*. The *Queue* itself is constructed as an Abstract Data Type.



PSPEC 4: Process Load Update

In *Process Load Update*, a new *Load* received from the Outgoing Processor is placed into an appropriate IE which is then transferede in a *Network Status Message*.

```
Inputs:
    Load: Real
Outputs:
    Network_Status_Msg: Network Status Indication Message
Operation:
    1. DECLARE Load_IE: Network Load IE
    2. Load_IE := ConstructIE_N_Load (Load)
    3. Network_Status_Msg := ConstructMsg_N_Status_Ind (Load_IE)
    4. STOP
```

PSPEC 5: Process Reject Message

Process Reject Message must deal with messages that are flushed from the Outgoing Processor. In the case of an *End System*, these messages are dropped, but when not in an *End System,* they are converted back into Indications and sent as a *Network Data Message* for re-routing.

```
Inputs:
    Reject_Data_Msg: Network Data Request Message
    End_System: Boolean
Outputs:
    Network_Data_Msg: Network Data Indication Message
Operation:
    1. IF End_System = True THEN
        1. Network_Data_Msg :=
                   ConvertMsg_N_Data_Req_To_Ind (Reject_Data_Msg)
    2. STOP
```

PSPEC 1.1: Classify Datalink Message

The message must be classified according to its type so that it can be processed by the appropriate task. This is done by looking at the type of the message.

```
Inputs:
    Datalink_Msg: Message
Outputs:
    Connect_Msg: Datalink Connect Indication Message
    Disconnect_Msg: Datalink Disconnect Indication Message
    Status_Msg: Datalink Status Indication Message
    Data_Msg: Datalink Data Indication Message
Operation:
    1. If Type (Connect_Msg) = Type (Datalink_Msg) THEN
        1. Connect_Msg := Datalink_Msg
    2. If Type (Disconnect_Msg) = Type (Datalink_Msg) THEN
        1. Disconnect_Msg := Datalink_Msg
    3. If Type (Status_Msg) = Type (Datalink_Msg) THEN
        1. Status_Msg := Datalink_Msg
    4. If Type (Data_Msg) = Type (Datalink_Msg) THEN
        1. Data_Msg := Datalink_Msg
    5. STOP
```

PSPEC 1.2: Process Data Message

In *Process Data Message*, the received Datalink *Data Message* must have its content extracted, after which the embedded *Network Data Message* is extracted and examined. If we are an *End System* and the Address corresponds to our *Address*, then the message is accepted. If we are not an *End System*, then the message is always accepted. An accepted message is translated into an Indication before being sent as a *Network Data Message*.

```
Inputs:
    Data_Msg: Datalink Data Indication Message
    End_System: Boolean
    Address: Integer
Outputs:
    Network_Data_Msg: Network Data Indication Message
Operation:
    1. DECLARE Req_Msg: Network Data Request Message
    2. Req_Msg := ExtractMsg_DL_Data_Ind (Data_Msg)
    3. IF End_System = True THEN
        1. DECLARE Dest_Address: Integer
        2. Dest_Address :=
                   ExtractMsg_N_Data_Req (Req_Msg, DESTADDRESS)
        3. IF Dest_Address != Address THEN
            1. STOP
    4. Network_Data_Msg := ConvertMsg_N_Data_Req_To_Ind (Req_Msg)
    5. STOP
```

PSPEC 1.3: Process Connect Message

In *Process Connect Message*, the *Connect Message* will trigger the *Start* activation for the Outbound processing, along with the generation of a *Network Connect Message* for the Upper Layer.

```
Inputs:
    Connect_Msg: Datalink Connect Indication Message
Outputs:
    Start: SIGNAL
    Network_Connect_Msg: Network Connect Indication Message
Operation:
    1. SIGNAL Start
    2. Network_Connect_Msg := ConstructMsg_N_Connect_Ind ()
    3. STOP
```

PSPEC 1.4: Process Disconnect Message

In *Process Disconnect Message*, the *Disconnect Message* will trigger the *Stop* activation for the Outbound processing, along with the generation of a *Network Disconnect Message* for the Upper Layer.

```
Inputs:
    Disconnect_Msg: Datalink Disconnect Indication Message
Outputs:
    Stop: SIGNAL
    Network_Disconnect_Msg: Network Disconnect Indication Message
Operation:
    1. SIGNAL Stop
    2. Network_Disconnect_Msg := ConstructMsg_N_Disconnect_Ind ()
    3. STOP
```

PSPEC 1.5: Process Status Message

In *Process Status Message*, the *Status Message* will trigger the *Release* activation for the Outbound processing only if the extracted content is a released *Datalink Flow Control IE*.

```
Inputs:
    Status_Msg: Datalink Status Indication Message
Outputs:
    Release: SIGNAL
Operation:
    1. DECLARE Flow_IE: Datalink Flow Control IE
    2. DECLARE Flow_State: Boolean
    3. Flow_IE := ExtractMsg_N_Status_Ind (Status_Msg)
    4. Flow_State := ExtractIE_DL_Flow_Control (Flow_IE)
    5. IF Flow_State = False THEN
        1. SIGNAL Release
    6. STOP
```

PSPEC 3.1: Initialise Queue

When activated, *Initialise Queue* will create the instance of the Queue ADT using appropriate configuration (policy and size). The *Wait for Release* flag is also set.

```
Inputs:
    Start: SIGNAL
    Queue_Policy: String
    Queue_Size: Integer
Outputs:
    Update_Status: SIGNAL
    Wait_For_Release: Boolean
    Queue_Index: Integer
Operation:
    1. Wait_For_Release := False
    2. Queue_Index := Queue_Create (Queue_Policy, Queue_Size)
    3. SIGNAL Update_Status
    4. STOP
```

PSPEC 3.2: Flush Queue

Flush Queue is called when the Network Layer has stopped, so that all content from the queue is extracted and passed out as *Reject Data Message*. The *Wait for Release* and *Queue* is updated.

```
Inputs:
    Stop: SIGNAL
    Queue_Index: Integer
Outputs:
    Wait_For_Release: Boolean
    Reject_Data_Msg: Network Data Request Message
    Update_Status: SIGNAL
Operation:
    1. Wait_For_Release := False
    Label_Loop_Next:
    2. IF Queue_Size (Queue_Index) > 0 THEN
        1. Reject_Data_Msg := Queue_Extract (Queue_Index)
        2. __output__
        3. GOTO Label_Loop_Next
    3. Queue_Index := Queue_Destroy (Queue_Index)
    4. SIGNAL Update_Status
    5. STOP
```

PSPEC 3.3: Release Queue

Release Queue is called when the next item in the *Queue* can be sent, however it may be the case that the *Queue* is empty, so the *Wait for Release* flag will be set. The resultant message, if any, is sent as an *Outgoing Data Message*.

```
Inputs:
    Release: SIGNAL
    Queue_Index: Integer
Outputs:
    Wait_For_Release: Boolean
    Outgoing_Data_Msg: Network Data Request Message
    Update_Status: SIGNAL
Operation:
    1. IF Queue_Size (Queue_Index) = 0 THEN
        1. Wait_For_Release := False
        2. STOP
    2. Outgoing_Data_Msg := Queue_Extract (Queue_Index)
    3. Wait_For_Release := True
    4. SIGNAL Update_Status
    5. STOP
```

PSPEC 3.4: Insert Queue

Insert Queue is called when an *Arrived Data Message* must be placed into the *Queue*. This action is carried out, and may result in the immediate transmission of an *Outgoing Data Message* if the *Queue* is empty.

```
Inputs:
     Arrived_Data_Msg: Network Data Request Message
     Wait_for_Release: Boolean
     Queue_Index: Integer
 Outputs:
     Wait_For_Release: Boolean
     Outgoing_Data_Msg: Network Data Request Message
     Update_Status: SIGNAL
 Operation:
     1. IF Queue_Size (Queue_Index) = Queue_Length (Queue_Index) THEN
         1. STOP
     2. IF Wait_For_Release = False THEN
         1. Outgoing_Data_Msg := Arrived_Data_Msg
         2. Wait_For_Release := True
         3. STOP
     3. Queue_Insert (Queue_Index, Arrived_Data_Msg)
     4. SIGNAL Update_Status
     5. STOP
```

PSPEC 3.5: Indicate Load

When *Indicate Load* is called, it will provide a normalised size of the *Queue* between 0 and 1. It does this by looking at the ratio of the *Size* to the *Length*.

```
Inputs:
    Update: SIGNAL
    Queue_Index: Integer
Outputs:
    Load: Real
Operation:
    1. Load := Queue_Size (Queue_Index) / Queue_Length (Queue_Index)
    2. STOP
```

ADT 3. Queue

<u>Overview</u>

The Queue ADT is provided in order to supply a basic FIFO Queue functionality with additional support for specific policies relating to the insertion and deletion of items. There are three main building blocks in the Queue ADT.

1. Queue Primitive Operations -- This consists of a "mini ADT" that only know how to store and retrieve opaque handles in a queue. These do not know anything about BONeS or policies, and are purposely constructed to hide complexity--they should be testable as a generic ADT.
2. Queue Context Primitive Operations -- This consists of primitives that know how to construct, deconstruct and store queue handle instances. These do not know

anything about BONeS or policies except that they retain a copy of the policy type (i.e. in a database role).

3. BONeS Operations -- This consists of the operations that were designed into the BONeS modules. These operations use the more primitive operations just mentioned, and do know how to deal with BONeS specifics.

Input and Output Policies

The Queue ADT is constructed with flexibility in terms of what policies should be enacted when inserting or extracting message from the queue (we are addressing the "upper" portion of the queue, that which has knowledge of BONeS). The current design has three policies for each:

Input Policy.

- *Drop Tail* -- When a message is to be inserted that overflows the queue, the message is discarded. This is an original and very simplistic policy, but has been shown to have significant problems as the result of much research.
- *Random Drop* -- When a message is to be inserted that overflows the queue, a message is randomly selected from within the queue and dropped. The new message is then inserted at the end of the queue. This mechanism has shown itself to be much fairer than Drop Trail, due to several reasons.
- *Random Early Detection (RED*) -- Floyd's RED mechanism monitors the average queue size and randomly discards messages if the size is above some threshold, even if the queue is not full. This allows transient traffic to still fill up the queue, but ensures that incipient congestion is indicated before it takes effect. *This was enventually not implemented.*

Output Policy.

- *Address Fair Queuing* -- A round robin approach is taken to select messages based on their address. The advantage of this approach is that it inherently strives towards allowing each address (or, flow) to have equivalent usage of the queue.
- *Size Priority* -- On every second extraction, an attempt is made to extract a message with a length lower than a specified threshold, the purpose of which is to give precedence to ACKs and as a side effect to lower sized interactive traffic. This attempts to cope with the ACK compression phenomena. Every second extraction is required otherwise starvation could occur [still can, to an extent].
- *Class Priority* -- IPv6 and other network protocols often include a class specification that can be used by intermediate systems in processing. This means that some classes, say interactive traffic, are given precedence over other classes, long transfers. In this case, every extraction attempts to locate the highest priority class. Starvation can occur in this scenario.

These policies are specified with the creation of an instance of the Queue, and subsequent "Insert" and "Extract" operations will use these disciplines.

Operations

Queue Primitive Operations.

Name	*Inputs*	*Outputs*	*Description*
_Queue_Create	Length	Queue	A Queue ADT is created with room for "Length" entries, and returned as the "Queue" handle.
_Queue_Destroy	Queue	Queue	The Queue ADT corresponding to the "Queue" handle is destroyed. Any entries still in the "Queue" are destroyed.
_Queue_Insert	Queue, Element	Queue, Result	The "Element" is inserted into the "Queue" if there is space and a True "Result" is returned and the "Queue" is modified. If there is no space, then a False "Result" is returned and the "Queue" remains unmodified.
_Queue_Get_Head	Queue	Queue, Element	If there are any entries in the "Queue", then the very first (FIFO discipline) "Element" is removed and returned, otherwise an invalid "Element" is returned.
_Queue_Get_Element	Queue, Offset	Queue, Element	If the size of the "Queue" is less than the "Offset", then the "Element" at the "Offset" is removed and returned, otherwise an invalid "Element" is returned.
_Queue_Peek_Element	Queue, Offset	Queue, Element	If the size of the "Queue" is less than the "Offset", then a copy of the "Element" at the "Offset" is returned, otherwise an invalid "Element" is returned.
_Queue_Get_Size	Queue	Size	The count of "Elements" in the "Queue" is returned as "Size".
_Queue_Get_Length	Queue	Length	The created "Length" of the "Queue" is returned.

Queue Context Operations.

Name	_QueueTable_Alloc (Policy, Length) --> Index
Descr.	A free entry in the Queue Table is located. A "Queue" instance is created using "_Queue_Create" of desired "Length"; this and the "Policy" is retained in the Table. The "Index" of this entry is returned.

PseudoCode	`Index := 0`
	`WHILE _Table[Index].Active = True`
	`    Index := Index + 1`
	`ENDWHILE`
	`_Table[Index].Queue := _Queue_Create (Length)`
	`_Table[Index].Policy := Policy`
	`_Table[Index].Active := True`

Name	_QueueTable_Free (Index) --> Index
Descr.	The entry in the Queue Table corresponding to "Index" is freed and returned; this is done using "_Queue_Destroy".

PseudoCode	`IF _Table[Index].Active = True THEN`
	`        _Queue_Destroy (_Table[Index].Queue)`
	`        _Table[Index].Active := False`
	`ENDIF`
	`Index := -1`

Name	_QueueTable_GetQueue (Index) --> Queue
Descr.	The "Queue" handle corresponding to "Index" is returned.
PseudoCode	`Queue := _Table[Index].Queue`

Name	_QueueTable_GetPolicy (Index) --> Policy
Descr.	The "Policy" information corresponding to "Index" is returned
PseudoCode	`Policy := _Table[Index].Policy`

BONeS Operations.

Name	Queue_Create (Policy, Length) --> Index
Descr.	Create a "Queue" with specified "Policy" and "Length". A warning is given if the "Queue" has already been created.

PseudoCode	`IF _QueueTable_GetQueue (Index) != Null THEN`
	`    ERROR "Queue is already created"`
	`ELSE`
	`    Index := _QueueTable_Alloc (Policy, Length)`
	`ENDIF`

Name	Queue_Destroy (Index) --> Index
Descr.	Destroy a "Queue". A warning is given if the "Queue" has not already been created.

PseudoCode	`IF _QueueTable_GetQueue (Index) = Null THEN`
	`    ERROR "Queue has already been destroyed"`
	`ELSE`
	`     Index := _QueueTable_Free (Index)`
	`ENDIF`

Name	Queue_Insert (Index, Element) --> Boolean
Descr.	Attempt to insert a new "Element" into the "Queue" according to the defined input policy. There are three input policies, Drop Tail, Drop Random and Random Early Detection (RED).

Pseud oCode

```
Queue := _QueueTable_GetQueue (Index)
IF Queue = NULL
    ERROR "Queue not initialised"
ELSE
    Policy := _QueueTable_GetPolicy (Index)

    CASE Policy:

        POLICY_DROP_TAIL:

            IF _Queue_Get_Size (Queue) < _Queue_Get_Length (Queue)
THEN
                _Queue_Insert (Queue, Element)
                Result := True
            ELSE
                Result := False
            ENDIF

        POLICY_DROP_RANDOM:

            IF _Queue_Get_Size (Queue) >= _Queue_Get_Length (Queue)
THEN
                Random := UNIFORM_RANDOM (0, _Queue_Get_Size
(Queue))
                _Queue_Get_Element (Queue, Random)
            ENDIF

            _Queue_Insert (Queue, Element)
            Result := True

        POLICY_RED:

            Result := False

    ENDCASE

ENDIF
```

Name Queue_Extract (Index) --> Element

Descr. Attempt to extract the next "Element" from the "Queue" according to the specified policy. This occurs by setting up a filter to indicate that all entries are valid, then successively removing entries in the filter according to defined policies. At the end, the first entry of those remaining will be used, but if none remain then the head of the queue will be used. The action for the specified filter operations is as such:

1. _Filter_On_Class -- remove all entries from the filter array other than those for the highest priority that is present in the array.

2. _Filter_On_Address -- remove all entries from the filter array other than those for the next address following the previously used address.

3. _Filter_On_Size -- if this is an even iteration, then remove all entries from filter array other than those that are equal to the entry with the smallest size.

PseudoCode

```
Queue := _QueueTable_GetQueue (Index)
IF Queue = Null THEN
    ERROR "Queue not Initialised"
ELSE
    IF _Queue_Get_Size (Queue) = 0 THEN
        Result := False
    ELSE
        Element_Array := Null
        Policy := _QueueTable_GetPolicy (Index)

        IF Policy & POLICY_OUT_ADDRESS THEN
            Element_Array := _Filter_On_Address (Element_Array)
        ENDIF

        IF Policy & POLICY_OUT_SIZE THEN
            Element_Array := _Filter_On_Size (Element_Array)
        ENDIF

        IF Policy & POLICY_OUT_CLASS THEN
            Element_Array := _Filter_On_Class (Element_Array)
        ENDIF

        IF SIZE (Element_Array) = 0 THEN
            Element := _Queue_Get_Head (Queue)
        ELSE
            Element := _Queue_Get_Element (Element_Array[0])
        ENDIF

        Result := True

    ENDIF

ENDIF
```

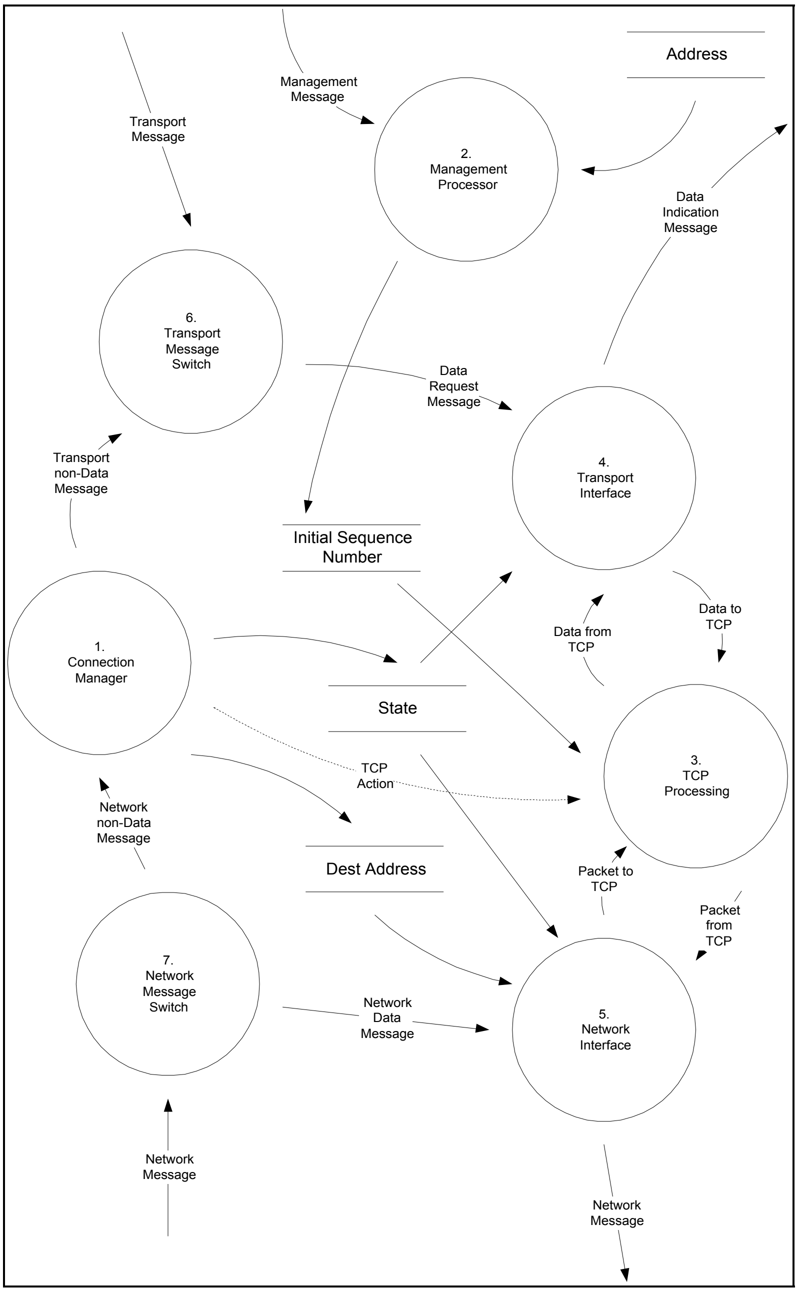
<table>
<tr><td><em>Name</em></td><td>Queue_Size (Index) --> Size</td></tr>
<tr><td><em>Descr.</em></td><td>Return the size of the "Queue".</td></tr>
<tr><td><em>PseudoCode</em></td><td><pre>
Queue := _QueueTable_GetQueue (Index)
IF Queue = Null THEN
    ERROR "Queue not initialised"
ELSE
    Size := _Queue_Get_Size (Queue)
ENDIF
</pre></td></tr>
</table>

<table>
<tr><td><em>Name</em></td><td>Queue_Length (Index) --> Length</td></tr>
<tr><td><em>Descr.</em></td><td>Return the length of the "Queue".</td></tr>
<tr><td><em>PseudoCode</em></td><td><pre>
Queue := _QueueTable_GetQueue (Index)
IF Queue = Null THEN
    ERROR "Queue not initialised"
ELSE
    Length := _Queue_Get_Length (Queue)
ENDIF
</pre></td></tr>
</table>

2.3. Transport Layer

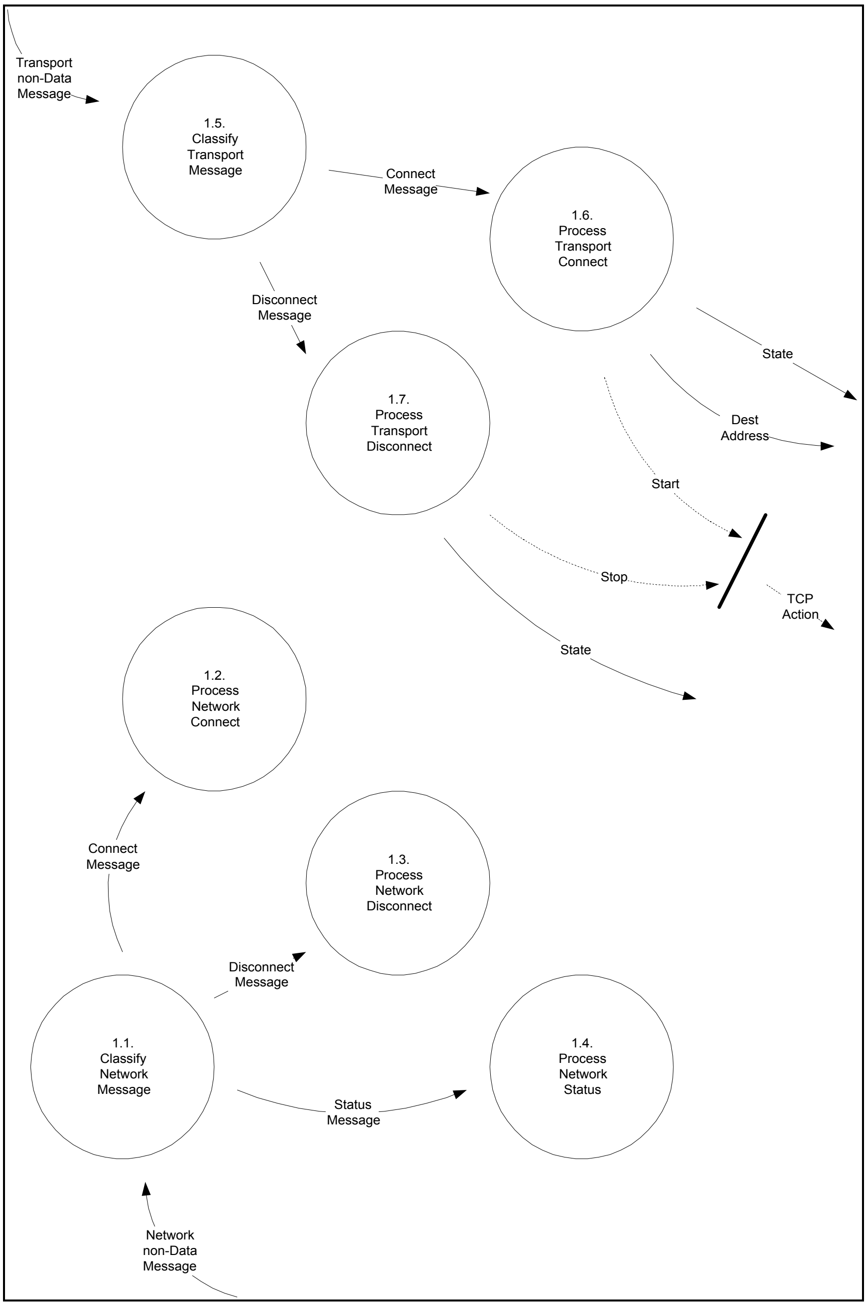
DFD 0: Top

In Top Level DFD, the *Transport Message Switch* and *Network Message Switch* ensure that *Data* and *non-Data Messages* are correctly routed. The *non-Data Messages* are processed by the *Connection Manager* and as a result the *State* of the current session may be changed, or its *Dest Address* may be set. *The Management Processor* uses the *Address* to processes a *Management Message* that may change *Initial Sequence Number*. *Data* Messages pass through either the *Transport Interface* or *Network Interface*. The core work is contained with *TCP Processing* which receives *Start* and *Stop* notifications from the *Connection Manager*.



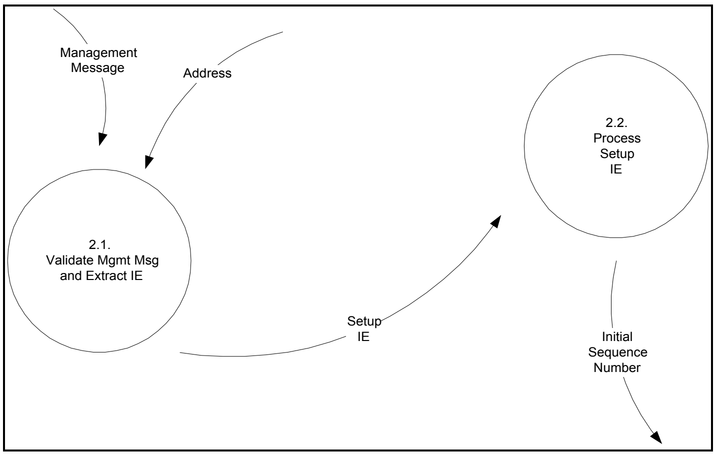
DFD 1: Connection Manager

The *Connection Manager* has two divides. The first is the processing of *Network non-Data Messages*. These are routed by *Classify Network Message* as *Connect Message*, *Disconnect Message* or *Status Message* and respectively processed *by Process Network Connect*, *Process Network Disconnect* or *Process Network Status*. *Transport non-Data Messages* are routed by *Classify Transport Message* to either *Process Transport Connect* or *Process Transport Disconnect*. Both result in changes to *State* and cause *Start* or *Stop* activations, respectively. A *Dest Address* is extracted in connect processing.


Transport non-Data Message
1.5. Classify Transport Message
Connect Message
1.6. Process Transport Connect
Disconnect Message
1.7. Process Transport Disconnect
State
Dest Address
Start
Stop
TCP Action
State
1.2. Process Network Connect
Connect Message
1.3. Process Network Disconnect
Disconnect Message
1.1. Classify Network Message
Status Message
1.4. Process Network Status
Network non-Data Message

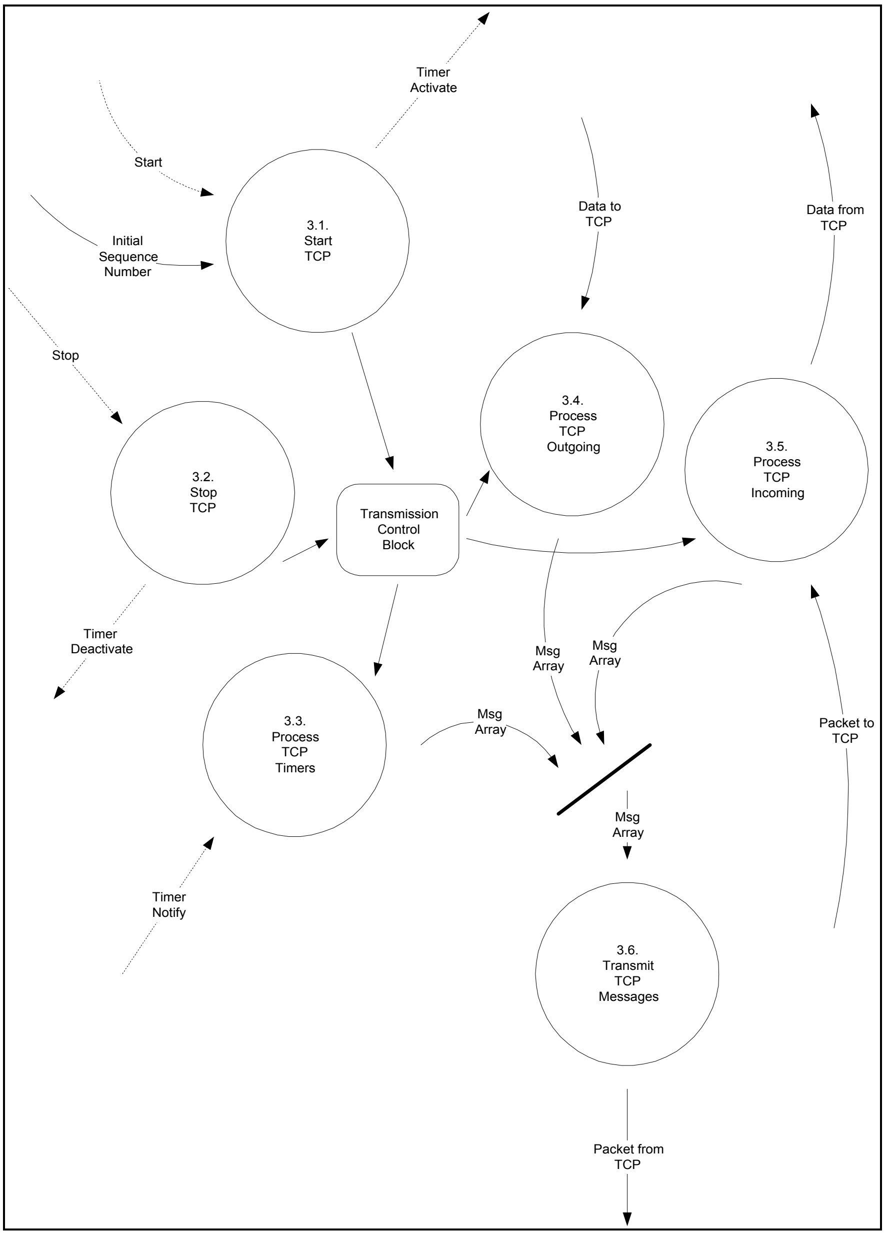
DFD 2: Management Processor

The *Management Processor* has a straight forward partitioning. Firstly, the message is validated in *Validate Mgmt Message and Extract IE* to ensure that it has the correct destination address and content, after which the content is processed according its type. The only specific content processed at this point in time is the *Transport Setup IE* in *Process Setup IE* -- this results in a change to the *Initial Sequence Number*.



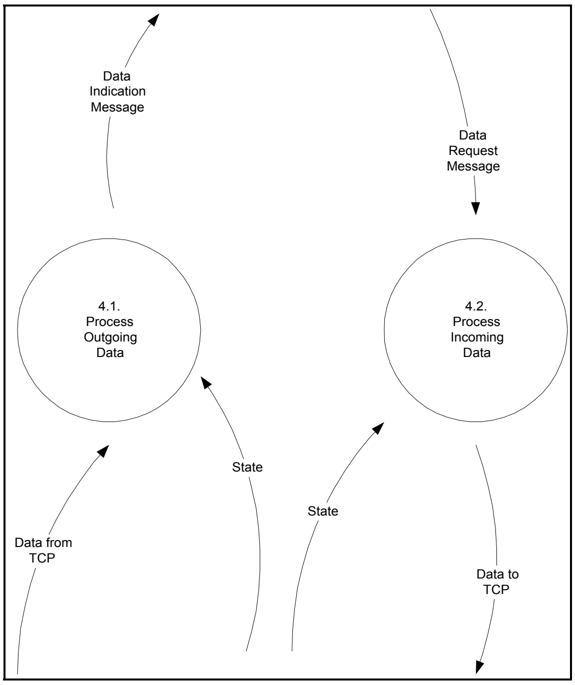
DFD 3: TCP Processing

In *TCP Processing*, the *Start* indication is used to activate *Start TCP* which creates a *TCB* with *Initial Sequence Number* and thence activates the timer. Periodic *Timer Notifies* are processed by *Process TCP Timer*, whilst *Data to TCP* is dealt with in *Process TCP Outgoing* and *Packet to TCP* in *Process TCP Incoming* (this may result in *Data from TCP*). The three central processing functions may all output *Msg Array* from which each Msg is extracted in *Transmit TCP Messages* as a *Packet from TCP*. When *Stop* is activated, the *TCB* is destroyed, and the timer is deactivated. Note that the (*TCB*) *Transmission Control Block* is used by all processes but is not connected to them all to reduce complexity in the diagram. The use of a *Msg Array* was the result of an iteration back from implementation.



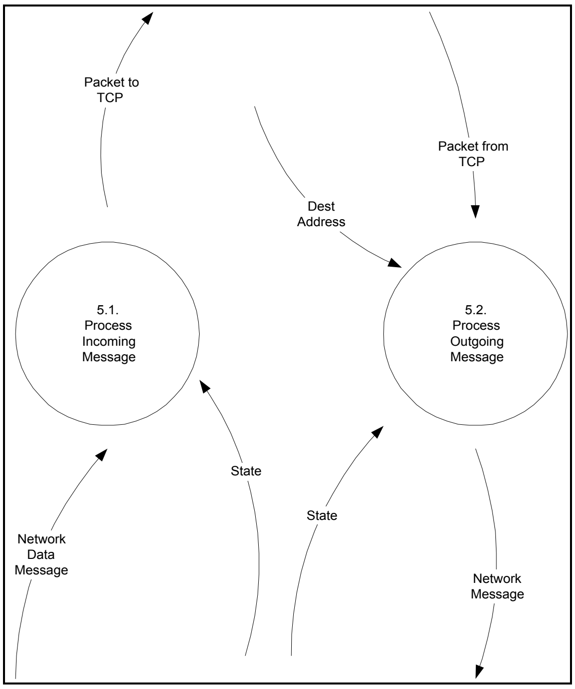
DFD 4: Transport Interface

The *Transport Interface* is concerned with mapping between Transport Messages and raw data as processed by the transport protocol. Arriving *Data Request Messages* are dealt with in *Process Incoming Data* which results in *Data to TCP*. *Arriving Data from TCP* is dealt with in *Process Outgoing Data* and results in a *Data Indication Message*. Note that in both cases, the *State* is used to ensure that the Transport Session is active before processing occurs.



DFD 5: Network Interface

The *Network Interface* is concerned with mapping between Network Messages and internal End-to-End "packets" as used by the transport protocol. Arriving *Network Data Messages* are dealt with in *Process Incoming Message* which results in a *Packet to TCP*. An arriving *Packet from TCP* is dealt with in *Process Outgoing Message* and results in a *Network Message.* Note that in both cases, the *State* must indicate that the Transport Session is active, or there will be no processing.



PSPEC 6: Transport Message Switch

The *Transport Message Switch* takes *Transport Messages* and categorises them into either *Transport Data Messages* or *Transport non-Data Messages*.

```
Inputs:
    Transport_Msg: Transport Message
Outputs:
    Transport_Data_Msg: Transport Data Request Message
    Transport_Non_Data_Msg: Transport Message
Operation:
    1. IF Type (Transport_Msg) = Type (Transport_Data_Msg)
        1. Transport_Data_Msg := Transport_Msg
        2. STOP
    2. Transport_Non_Data_Msg := Transport_Msg
    3. STOP
```

PSPEC 7: Network Message Switch

The *Network Message Switch* takes *Network Messages* and categorises them into *either Network Data Messages* or *Network non-Data Messages*.

```
Inputs:
    Network_Msg: Network Message
Outputs:
    Network_Data_Msg: Network Data Indication Message
    Network_Non_Data_Msg: Network Message
Operation:
    1. IF Type (Network_Msg) = Type (Network_Data_Msg)
        1. Network_Data_Msg := Network_Msg
        2. STOP
    2. Network_Non_Data_Msg := Network_Msg
    3. STOP
```

PSPEC 1.1: Classify Network Message

The message must be classified according to its type so that it can be processed by the appropriate task. This is done by looking at the type of the message.

```
Inputs:
    Network_Msg: Message
Outputs:
    Connect_Msg: Network Connect Indication Message
    Disconnect_Msg: Network Disconnect Indication Message
    Status_Msg: Network Status Indication Message
Operation:
    1. If Type (Connect_Msg) = Type (Network_Msg) THEN
        1. Connect_Msg := Network_Msg
    2. If Type (Disconnect_Msg) = Type (Network_Msg) THEN
        1. Disconnect_Msg := Network_Msg
    3. If Type (Status_Msg) = Type (Network_Msg) THEN
        1. Status_Msg := Network_Msg
    4. STOP
```

PSPEC 1.2: Process Network Connect

The message is currently not processed.

```
Inputs:
    Connect_Msg: Network Connect Indication Message
Outputs:
    n/a
Processing:
    1. STOP
```

PSPEC 1.3: Process Network Disconnect

The message is currently not processed.

```
Inputs:
    Disconnect_Msg: Network Disconnect Indication Message
Outputs:
    n/a
Processing:
    1. STOP
```

PSPEC 1.4: Process Network Status

The message is currently not processed.

```
Inputs:
    Status_Msg: Network Status Indication Message
Outputs:
    n/a
Processing:
    1. STOP
```

PSPEC 1.5: Classify Transport Message

The message must be classified according to its type so that it can be processed by the appropriate task. This is done by looking at the type of the message.

```
Inputs:
    Transport_Msg: Message
Outputs:
    Connect_Msg: Transport Connect Indication Message
    Disconnect_Msg: Transport Disconnect Indication Message
Operation:
    1. If Type (Connect_Msg) = Type (Transport_Msg) THEN
        1. Connect_Msg := Transport_Msg
    2. If Type (Disconnect_Msg) = Type (Transport_Msg) THEN
        1. Disconnect_Msg := Transport_Msg
    3. STOP
```

PSPEC 1.6: Process Transport Connect

The message is used to obtain the *Dest Address* for the Session, update the known *State* of the Session and thence to *Start* the Session's processing.

```
Inputs:
    Connect_Msg: Transport Connect Request Message
Outputs:
    State: Boolean
    Start: SIGNAL
    Dest_Address: Integer
Processing:
    1. Dest_Address := ExtractMsg_T_Connect_Req (Connect_Msg)
    2. State := True
    3. SIGNAL Start
    4. STOP
```

PSPEC 1.7: Process Transport Disconnect

The message is used to update the known *State* of the Session and thence to *Stop* the Session's processing.

```
Inputs:
    Disconnect_Msg: Transport Disconnect Request Message
Outputs:
    State: Boolean
    Stop: SIGNAL
Processing:
    1. State := False
    2. SIGNAL Stop
    3. STOP
```

PSPEC 2.1: Validate Mgmt Msg and Extract IE

The *Management Message* is inspected to ensure that is destined for this module, and that the content is valid via *Validate Mgmt Message and Extract IE*. The appropriate output is generated depending on the type of content.

```
Inputs:
    Mgmt_Msg: Management Set Indication Message
    Address: Integer
Outputs:
    Setup_IE: Transport Setup IE
Processing:
    1. DECLARE Unknown_IE: IE
    2. IF Address (Mgmt_Msg) = Address THEN
        1. Unknown_IE := ExtractMsg_M_Set_Ind (Mgmt_Msg)
        2. IF Type (Unknown_IE) = Type (Setup_IE)
            1. Setup_IE := Unknown_IE
    3. STOP
```

PSPEC 2.2: Process Setup IE

The content of the *Setup IE* is processed in *Process Setup IE* and used to update the internal *Initial Sequence Number*.

```
Inputs:
    Setup_IE: Transport Setup IE
Outputs:
    Initial_Sequence_Number: Integer
Processing:
    1. Initial_Sequence_Number := ExtractIE_T_Setup (Setup_IE)
    2. STOP
```

PSPEC 4.1: Process Outgoing Data

Process Outgoing Data is concerned with taking *Data from TCP* and encapsulating it within a *Data Indication Message*, but this will only occur if the *State* is active.

```
Inputs:
    Data_from_TCP: Integer
    State: Boolean
Outputs:
    Data_Indication_Msg: Transport Data Indication Message
Processing:
    1. IF State = True THEN
        1. DECLARE Data_Msg: Application Data Message
        2. Data_Msg := ConstructMsg_Applic_Data (Data_from_TCP)
        3. Data_Indication_Msg := ConstructMsg_T_Data_Req (Data_Msg)
    2. STOP
```

PSPEC 4.2: Process Incoming Data

Process Outgoing Data is concerned with taking a *Data Request Message* from TCP and converting it into *Data to TCP*, but this will only occur if the *State* is active.

```
Inputs:
    Data_Request_Msg: Transport Data Request Message
    State: Boolean
Outputs:
    Data_to_TCP: Integer
Processing:
    1. IF State = True THEN
        1. DECLARE Data_Msg: Application Data Message
        2. Data_Msg := ExtractMsg_T_Data_Req (Data_Request_Msg)
        3. Data_to_TCP := ExtractMsg_Applic_Data (Data_Msg)
    2. STOP
```

PSPEC 5.1: Process Incoming Message

Process Incoming Message is concerned with taking a *Network Data Message* from the Network Layer and extracting the *Packet to TCP*, but this will only occur if the *State* is active.

```
Inputs:
    Network_Data_Msg: Network Data Indication Message
    State: Boolean
Outputs:
    Packet_to_TCP: TCP Packet
Processing:
    1. IF State = True THEN
        1. Packet_to_TCP := ExtractMsg_N_Data_Ind (Network_Data_Msg)
    2. STOP
```

PSPEC 5.2: Process Outgoing Message

Process Outgoing Message is concerned with taking a *Packet from TCP* and constructing a *Network Message* from it, but this will only occur if the *State* is active. The correct *Dest Address* is also placed into the created message.

```
Inputs:
    Packet_from_TCP: TCP Packet
    Dest_Address: Integer
    State: Boolean
Outputs:
    Network_Msg: Network Data Request Message
Processing:
    1. IF State = True THEN
        1. Network_Msg :=
            ConstructMsg_N_Data_Req (Dest_Address, Packet_from_TCP)
    2. STOP
```

PSPEC 3.1: Start TCP

Start TCP activates due to a Transport Session *Start* indication, and initialises TCP for processing. This initialisation consists of allocating a Transmission Control Block using *TCP Create* and using the *Initial Sequence Number* to configure the TCB. The TCP Timer is also activated, it runs every 100ms.

```
Inputs:
    Start: SIGNAL
    Initial_Sequence_Number: Integer
Outputs:
    Timer_Activate: SIGNAL
    TCB_Index: Integer
Processing:
    1. TCB_Index := TCP_Create (Initial_Sequence_Number)
    2. Timer_Activate := SET_TIMER (100MS)
    3. STOP
```

PSPEC 3.2: Stop TCP

Stop TCP activates due to a Transport Session *Stop* indication, and stops TCP processing. This consists of de-allocating the currently used TCB and shutting down the TCP Timer.

```
Inputs:
    Stop: SIGNAL
    TCB_Index: Integer
Outputs:
    Timer_Deactivate: SIGNAL
Processing:
    1. TCB_Index := TCP_Destroy (TCB_Index)
    2. Timer_Deactivate := UNSET_TIMER ()
    3. STOP
```

PSPEC 3.3: Process TCP Timers

Process TCP Timers is activated by *Timer Notify* every 100ms. Its purpose is to execute any scheduled TCP activity (such as delayed acknowledgments, retransmission timeouts, persist timeouts, etc). It may result in the output of a *Msg Array*.

```
Inputs:
    Timer_Notify: SIGNAL
    TCB_Index: Integer
Outputs:
    Msg_Array: ARRAY OF TCP Packet
Processing:
    1. Msg_Array := TCP_Process_Timer (TCB_Index)
    2. STOP
```

PSPEC 3.4: Process TCP Outgoing

Process TCP Outgoing is concerned with taking *Data to TCP* and attempting to package it for end to end transmission. This may or may not happen immediately due to internal buffering that may occur, but it is possible for a *Msg Array* to result as an output.

```
Inputs:
    Data_to_TCP: Integer
    TCB_Index: Integer
Outputs:
    Msg_Array: ARRAY OF TCP Packet
Processing:
    1. Msg_Array := TCP_Process_Output (TCB_Index, Data_to_TCP)
    2. STOP
```

PSPEC 3.5: Process TCP Incoming

Process TCP Incoming must use the *Packet to TCP* to carry out receiver side TCP processing. The result of this is possibly *Data from TCP* to the Upper Layer, or *Msg Array* for the transmission of packets back to the peer.

```
Inputs:
    Packet_to_TCP: TCP Packet
    TCB_Index: Integer
Outputs:
    Data_from_TCP: Integer
    Msg_Array: ARRAY OF TCP Packet
Processing:
    1. (Msg_Array, Data_from_TCP) :=
         TCP_Process_Input (TCB_Index, Packet_to_TCP)
    2. STOP
```

PSPEC 3.6: Transmit TCP Messages

Due to an implementation concern, the originating messages from TCP processing are contained within a *Msg Array*. Each message in this array is extracted and sent via *Transmit TCP Messages* -- resulting in a number of *Packet from TCP* being output.

```
Inputs:
    Msg_Array: ARRAY OF TCP Packet
Outputs:
    Packet_from_TCP: TCP Packet
Processing:
    1. DECLARE Count: Integer
    2. Count := 0
    Label_Loop_Next:
    3. IF Count < Length (Msg_Array) THEN
        1. Packet_from_TCP := Msg_Array [Count]
        2. __output Packet_from_TCP
        3. Count := Count + 1
        4. GOTO Label_Loop_Next
    4. STOP
```

MODULE 3. Transmission Control Protocol

Overview

The Transmission Control Protocol (TCP) is moderately complex in design and implementation. As noted, our model of this protocol takes the ESTABLISHED processing phase and does not concern itself with the initialisation and termination processing. Our model is based upon BSD4.4/Net 3. There are a number of reasons for this:

1. It provides a conceptual and proven architecture to work from.
2. Although it aggregates much functionality, it does have a straightforward procedural manner and is easy to understand.
3. It contains enhancements beyond the TCP specifications and other implementations.

4. It is the primary platform used in research circles; therefore it is an appropriate testbed for comparative and explanatory purposes.
5. It has neatly separated TCP processing from other networking (and, for that matter, kernel) elements.
6. It ensures that the behaviour we see is actually that of a legitimate TCP implementation as opposed to something we may have done ourselves.

The design and implementation of our TCP processing was carried out from scratch, it does not contain anything from BSD4.4/Net 3, however its organisation is strongly mirrored.

The approach taken here is to construct the TCP protocol as a separable element, and to then build interface functions to allow it to communicate with BONeS. These interface functions are described first. From interface to internal, there are three elements to the core TCP processing: input processing, output processing and timer processing.

Input processing takes a TCP packet received from a peer TCP entity and carries out a set of procedures to verify the packet, and then to process specific aspects of it. The check involves ensuring that the data content of the packet is within our receive window; and processing involves round-trip-time computation, acknowledgment processing--including duplicate acknowledgments which trigger fast-retransmits--, data processing and window updating.

Output processing is concerned with taking data as supplied by the Upper Layer, checking whether it is possible (due to current conditions) to output a TCP packet, and then actually doing so. These checks involve examination of both TCP and congestion windows, outstanding ACKs, silly window conditions, and so on. This output processing may occur directly as a result of data supplied by the application, or due to timer and acknowledgment processing.

Timer processing is activated at regular periods and concerns itself with sending delayed acknowledgments, checking for retransmission timeouts and window probing. Round-Trip-Time and idle time parameters are also updated. Internally, either input or timer processing may result in the occurrence of output processing.

The operation of these three main functions requires a database of state information, referred to as the Transmission Control Block (TCB). It is initialised at the establishment of a conversation, and removed at termination. In addition, the TCP packets are defined as data structures that must correlate to those used in BONeS.

The proceeding sections intend to outline the design for the TCP processing. The design is not specific in giving pseudo-code, as the implementation resulted from mapping textual description with current BSD4.4/Net3 implementation into a rough, then refined implementation.

External Interface

Externally, there are six functions that are accessible. With this is a message structure used for the transfer of data between TCP end points.

Data

Messages

TCP Messages are used to deliver control and data information between TCP peers. The message structure, in the real world, is well defined. For the purposes of modelling, we use some fields from the defined message both as they are, and modified; along with additional fields to aid our simulations. The message contains:

Name	*BSD4.4 / Net 3 Name*	*Description / Reason for Inclusion*
Length	null	For computation of delivery time, IP length is used in real
Sequence Number	th_seq (32 bit)	Sequence number for data in the segment
Acknowledgment	th_ack (32 bit)	Acknowledgment for previous data received
Window	th_win (16 bit)	Current advertised window
Ack Flag	th_flags & TH_ACK	Whether segment does acknowledge previously recvd
Timestamp Flag	TF_RCVD_TSTMP	Whether timestamp is present
Recent Time	ts_val (32 bit)	Value of the time stamp
Time Now	ts_ecr (32 bit)	Value of the timestamp reply

The following fields were left out, for the given reasons:

Name	*BSD4.4 / Net 3 Name*	*Description / Reason for Exclusion*
Source Port	th_sport (16 bit)	We use Addresses for one to one mapping of associations
Dest Port	th_dport (16 bit)	As with Source Port
Data Offset	th_off (4 bit)	Header is always a fixed size
Flags	th_flags (8 bit)	We only use one flag, the Acknowledgment, which already has a field defined
Checksum	th_sum (16 bit)	Are not concerned with modelling errors
Urgent Offset	th_urp (16 bit)	We do not model urgent data transfer, as it can be considered normal data transfer

Functions

Name	TCP_Create (Initial Sequence Number) --> Index
Descr.	Create the instance of TCP. This produces an Integer index to be used in the following TCP functions. The Initial Sequence Number is used to set up send and receive sequence numbers.

Pseudo Code	`Index := TCB_Create ()`
	`Tcb := TCB_Lookup (Index)`
	`Init_Process (Tcb, Initial Sequence Number)`

Name	TCP_Destroy (Index)
Descr.	Destroy the instance of TCP.
Pseudo Code	`TCB_Destroy (Index)`

Name	TCP_Process_Timer (Index) --> BONeS_TCP_Msg
Descr.	Locate the TCB and call timer processing. There may be messages output from here.

Pseudo Code	`Tcb := TCB_Lookup (Index)`
	`Timer_Process (Tcb)`
	`Label_Get_Next:`
	`Msg := Get_From_Message_Queue ()`
	`IF Msg != Null THEN`
	`    BONeS_TCP_Msg := Convert_Msg_To_BONeS_Msg (Msg)`
	`    OUTPUT BONeS_TCP_Msg`
	`    GOTO Label_Get_Next`
	`ENDIF`

Name	TCP_Process_Output (Index, Data_Length) --> BONeS_TCP_Msg
Descr.	Insert the Data onto the outgoing queue, and then call the output processing. There may be messages output from here.

Pseudo Code	`Tcb := TCB_Lookup (Index)`
	`Tcb->Outgoing_Buffer := Tcb->Outgoing_Buffer + Data_Length`
	`Output_Process (Tcb)`
	`Label_Get_Next:`
	`Msg := Get_From_Message_Queue ()`
	`IF Msg != Null THEN`
	`    BONeS_TCP_Msg := Convert_Msg_To_BONeS_Msg (Msg)`
	`    OUTPUT BONeS_TCP_Msg`
	`    GOTO Label_Get_Next`
	`ENDIF`

Name	TCP_Process_Input (Index, BONeS_TCP_Msg) --> BONeS_TCP_Msg
Descr.	Convert the message into an internal representation, and then call input processing. There may be messages output from here.

Pseudo Code	`Tcb := TCB_Lookup (Index)`
	`Msg := Convert_BONeS_Msg_To_Msg (BONeS_TCP_Msg)`
	`Input_Process (Tcb, Msg)`
	`Label_Get_Next:`
	`Msg := Get_From_Message_Queue ()`
	`IF Msg != Null THEN`
	`    BONeS_TCP_Msg := Convert_Msg_To_BONeS_Msg (Msg)`
	`    OUTPUT BONeS_TCP_Msg`
	`    GOTO Label_Get_Next`
	`ENDIF`

Internal Functionality

The internally functionality consists of two significant aspects. The first being the data that is used in the functionality, and the second being the procedures used in processing that data.

Data

Transmission Control Block (TCB)

The Transmission Control Block (TCB) contains all necessary state variables for an instance of TCP processing. It has the following entries:

Name	*BSD4.4 / Net3 Name*	*Description*
Delayed Ack Flag	`t_flags & TF_DELACK`	Indicates whether Delayed Ack is scheduled
Ack Flag	`t_flags &`	Indicates whether Ack needs to be sent

	TF_ACKNOW	back to the remote
Persist Timer	t_timer & TCPT_PERSIST	Count down until persist activity should occur
Retransmit Timer	t_timer & TCPT_REXMT	Count down until retransmission is timed out
Send Window	snd_wnd	Current send window
Send Unacknowledged	snd_una	First unacknowledged sequence number
Send Next	snd_nxt	Next sequence number to send
Send Lower Window	snd_wl1	Lower edge of Send Window
Send Upper Window	snd_wl2	Upper edge of Send Window
Send Max	snd_max	Highest sequence number sent
Receive Window	rcv_wnd	Current receive window
Receive Next	rcv_nxt	Next expected receive sequence number
Send Congestion Window	snd_cwnd	Current window limitation due to congestion
Send Slow Start Threshold	snd_ssthresh	Point at which linear window increase kicks in
Idle	t_idle	Amount of time that TCP has been idle
Round Trip Time	t_rtt	Currently known/estimated round trip time
Round Trip Time Sequence Number	t_rttseq	Sequence number being used for round trip time estimation
Smoothed Round Trip Time	t_srtt	Round trip time after being smoothed due to inherent fluctuation
Round Trip Time Variance	t_rttvar	Variance occurred in round trip time
Round Trip Time Minimum	t_rttmin	Smallest value of the round trip time seen so far
Maximum Send Window	max_sndwnd	Maximum send window that occurred

Timestamp Flag	t_flags & TF_RCVD_TSTMP	Whether timestamp flags are in effect
Timestamp Recent	ts_recent	Last recent timestamp received
Timestamp Recent Age	ts_recent_age	When the timestamp was received
Time Now	t_now	Current virtual clock time; increased every 500ms
Retransmit Shift	t_rxtshift	Current backoff index for retransmit/persist
Retransmit Current	t_rxtcur	Current backoff value of retransmit/persist
Duplicate Acks	t_dupacks	Count of duplicate Acks that have been received
Maximum Segment	t_maxseg	The maximum size of a segment that we can send
Last Acknowledgment Sent	last_ack_sent	The last acknowledgment that we have sent.
Send Scale	snd_scale	Scaling used for the Send Window
Receive Scale	rcv_scale	Scaling used for the Receive Window
Timer Ticks	-	Used internally to count ticks and schedule fast or slow timer kicks
Allocated	-	Whether TCB is in used
Fragment Queue	seg_next, seg_prev	List of out of order fragments

The following TCP related state variables are left out, due to the given reasons:

Name	*BSD4.4 / Net3 Name*	*Description / Reason for Exclusion*
TCP State	t_state	We only have an ESTABLISHED state in our model
Force Output	t_force	We indicate whether forced directly when calling output processing
Flags	t_flags	We have explicit flags as opposed to a bit field

TCP Template	`t_template`	This is only used for performance reasons, we don’t need a header template
Send Urgent Pointer	`snd_urp`	We don’t have Urgent Data in our model
Initial Send Sequence Number	`iss`	We maintain the Initial Sequence Number outside of the TCB
Initial Receive Sequence Number	`irs`	We maintain the Initial Sequence Number outside of the TCB
Receive Urgent Pointer	`rcv_urp`	We don’t have Urgent Data in our model
Out of Band Data	`t_obbflags, t_iobc, t_softerror`	We don’t model out of band data.
Requested Scaling	`request_r_scale, requested_s_scale`	We have explicit Window Scaling available, there is no need to model the synchronisation

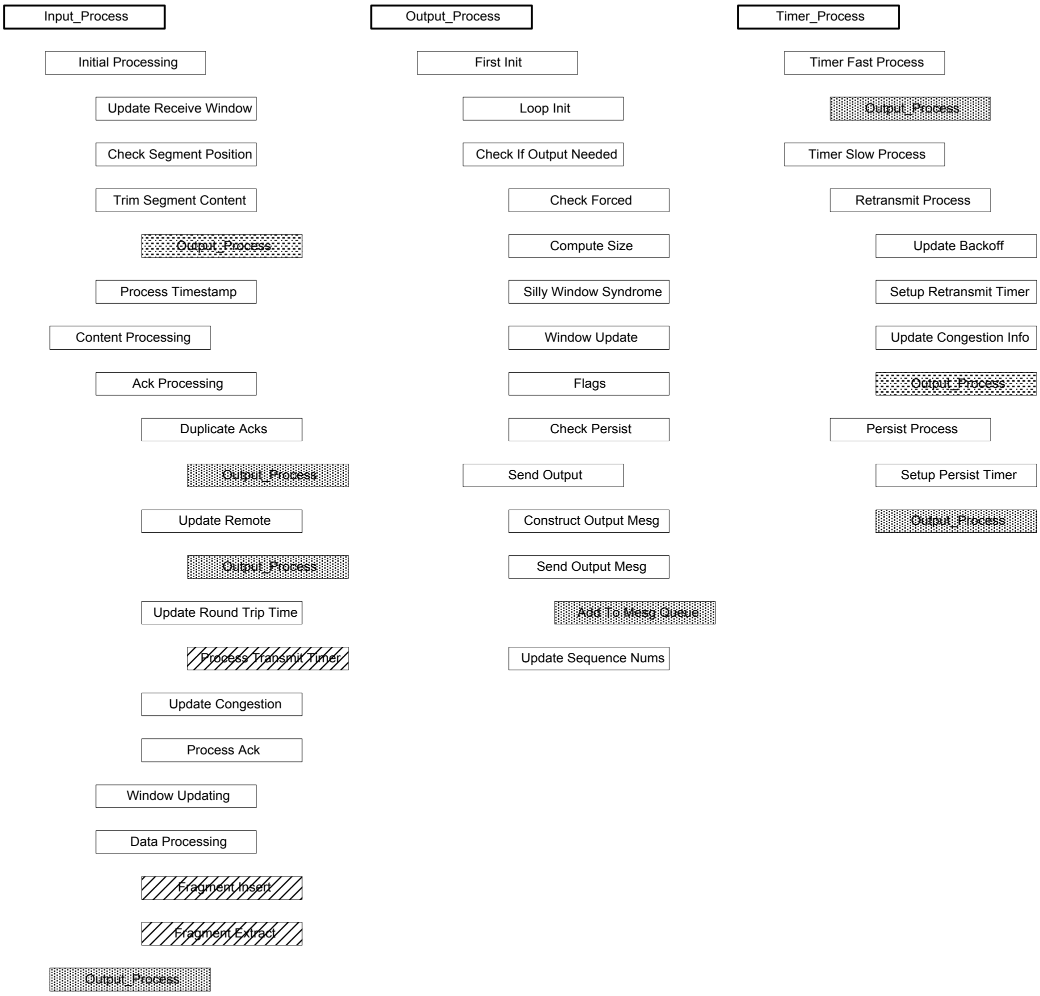
Ancillary Functions

Function Name	*Description*
TCB Create	There are no inputs. The function returns an index to a newly created TCB--that is stored in a global table. With this index, the TCP can subsequently execute using the TCB.
TCB Destroy	The input is the index of a previously created TCB. The TCB will then be destroyed, and subsequently the index becomes invalid and the TCB cannot be used.
TCB Lookup	The input is the index of a previously created TCB. The function will return a handle for the TCB for use in the TCP processing functions.
Convert BONeS_Message To Msg	A BONeS data structure, in its particular representation, is converted into an internal data structure for use in TCP processing. This occurs prior to Input processing.
Convert Msg To BONeS_Message	An internal data structure is converted into a BONeS data structure. This occurs after TCP processing as generated messages are given to BONeS.
Add To Message Queue	During operation of Output processing, a number of messages may result; they are queued temporarily until

	all processing has been completed. This function will queue the messages.
Get From Message Queue	The just mentioned messages are able to be dequeued; one at a time.
Fragment	TCP processing involves maintaining a queue of incoming fragments that arrive out of order. The Fragment module stores these and allows for the extraction of contiguous sections at the head of the queue. Hence, this module can be considered to provide the reassembly mechanisms.

TCP Processing Functions

In summary, the three significant processing functions are shown in the following call graph. Note the case of output processing being called from input and timer processing. The design is hierarchically and procedurally structured. Each of the three significant functions along with their respective internal functions are outlined.



1. Init_Process

The input to initialisation is the "Initial Sequence Number", and the "TCB". Each item in the TCB is then initialised to default values; which includes the following: No Delayed Ack or Ack flags; Timers set to zero; Send and Receive sequence numbers set to Initial Sequence Number; Send and Receive windows set to maximum; Congestion window and Slow-Start Threshold set to maximum; Round Trip Time values reset; Retransmit backoff value reset; Other miscellaneous variables reset.

2. Input_Process

The input here is a "Message" and the "TCB". What occurs is that the "Message" first has "Initial Processing" (2.1) applied, and then if the "Message" ACK flag is set, will have "Content Processing" (2.2) applied. It is possible that or subsequent to this, "Output_Process" (3) will be called to schedule data or acknowledgment output.

Function Name	*Description*
2.1. Initial Processing	The input stage requires some initial processing, this takes the form of carrying out several validity checks on the segment, and possibly tossing away the segment if any of these fail. The processing that occurs is: a. "Update Receive Window" -- Recompute the receive window [this is not affected by the incoming message, but it only needs to be done for use in input processing]. b. "Check Segment Position" -- Check the segment's position in the receive window, as it may need to be dropped if it lies outside the window. c. "Trim Segment Content" -- Cut out upper and lower chunks from the segment if they fall outside the window, note that this may also cause the entire segment to be dropped. d. "Process Timestamp" -- Extract the timestamp option from the segment and update the RTT. Before this, the idle flag is updated to indicate that the TCP is not idle any more.
2.1.1. Update Receive Window	Compute the value of the receive window based upon the current receive sequence numbers.
2.1.2. Check Segment Position	Check the segments position to see if it overlaps with the top of what has already been received, then remove that extraneous data from the segment.
2.1.3. Trim Segment Content	Check to see how much of the segment lies outside of the legitimate receive window: throw away partial or all of the data that overlaps. If the segment is entirely outside of the receive window, then drop it and send back an ACK to the peer to indicate so.
2.1.4. Process Timestamp	Extract the timestamp and related information from the segment, but do this only if the timestamp option is

	enabled, and the segment is a response to the last sent acknowledgment.
2.2. Content Processing	Process the content of a segment, taking several steps. These are the things that are done: a. "Ack Processing" -- Do all the things that occur when a segment is received with the ACK bit set. b. "Window Updating" -- Update the receive window based on the segment content. c. "Data Processing" -- Extract the content of the segment and do something with it; i.e. send it up to the upper layer or put it on the reassembly queue.
2.2.1. Ack Processing	Process lots of things in the input segment relating to segments when they have ACKs on them. The following is what is looked at: a. "Duplicate Acks" -- The reception of duplicate acks is used to fire up the "fast retransmit" mechanism of TCP that assumes that 3 such duplicate acks are a sign of lots segments. b. "Update Remote" -- Check to see how much data is acked, and more fundamentally, whether or not the ack is within the window. c. "Update Round Trip Time" -- This ACK may be coming back from a segment being timed, or alternatively use what is in the timestamp option. d. "Update Congestion" -- Must update the congestion window based on the incoming acks ("Ack clocking"). e. "Process Ack" -- Finally, the ACK is processed so that transmit buffer content is released and the appropriate sequence numbers are updated.
2.2.1.1. Duplicate Acks	This is where duplicate ACKs are processed. Increase the count of them until a threshold is reached, at which point scale back the slow start threshold and the congestion window then fire off TCP output as a guess that a packet has been dropped (but not picked up by the retransmit threshold). If more than the threshold of duplicate ACKs has been received, then pump up the congestion window by a segment so as to keep the pipe full : and then kick output processing.
2.2.1.2. Update Remote	If there are a lot of duplicate ACKs, may need to scale back the congestion window to the slow start threshold. Also, drop out here if the ACKs are for data that is above the window (should never happen ...).
2.2.1.3. Update Round Trip Time	Update the RTT estimators, taking into account two cases, the first being where there is a timestamp, so use this (much more reliable) information to do the RTT. Otherwise, if the ACK is greater than that which was sent out to time for this segment, use the estimated RTT.
2.2.1.4. Update Congestion	Update the congestion window, increase it just a tad but constrain it to the maximum window that can be sent.

2.2.1.5. Process Ack	Here, the ACK is actually used to slop out data from the transmit buffer; look at how much has been ACKed, and it either covers the whole buffer, or only part thereof. Note that in TCP, there are no selective acks, which kind of makes this process easier (at the cost of performance :-). Having finished updating the buffer, update the next and unacknowledged sequence number fields in the TCB.
2.2.2. Process Window Update	Process for a window update, by looking at the sent sequence numbers and the updated window. This is trying to make sure that window updates are only processed where the update is not an old one!
2.2.3. Process Data	Process the data that is in the segment. There are two cases (only for purposes of optimisation); the first is where receiving the next segment of data inline and there is nothing on the queue. For this, the data can be accepted straight away and passed up to the upper layer. The second case is where there are existing fragments, so we stick this into the reassembly queue and immediately attempt to extract anything that is at the head of the queue. Next, Setup a delayed ACK flag for the inline case, and a normal ACK for the other.

3. Output_Process

The complete output processing stage; initially, initialise some variables before we then loop around attempting to first check for output, and if there is a reason to output, then generate a segment and send it along with post-update of state.

Function Name	*Description*
3.1. Check If Output Needed	This is the first half of TCP output processing, which actually tries to determine whether or not something should be sent, and if so then establish the basic parameters (i.e. amount to send and so forth). If any check is true, then the second half of TCP output processing is called, if all checks fail then nothing occurs. The checks are as follows: a. "Check Forced" -- Special conditions that occur if output is being forced. b. "Compute Size" -- Determine how much data there is to send, within the constraints of window, buffer and other sizes. c. "Silly Window Syndrome" -- Check out the silly window syndrome conditions; these may or may not inhibit transmission. d. "Window Update" -- If sending a window update, check for it here. e. "Flags" -- certain specific flags; i.e. "ack", may require

	transmission. f. "Check Persist" -- finally, need to persist to probe for a window change.
3.1.1. Check Forced	Here, do processing that occurs only if forcing an output; remember the only condition for a forced output is during a window probe when persisting. So, ensure that _something_ is being sent, even if it is only a size of one. Also, the case may be that the window is not zero, therefore can kill the persist timer.
3.1.2. Compute Size	Here, figure out how much data can be sent. Firstly, compute the initial size as the minimum of the send buffer and the available window; from which subtract the amount that has already been sent in this window. After which; check for a negative length and check to see whether finished retransmitting. Finally, truncate the length to the maximum segment size that can be sent, and make a note to the effect that can come back here to send more.
3.1.3. Silly Window Syndrome	Silly Window Syndrome avoidance is carried out both by the send and receiver; here is the sender side of it. What occurs is that a set of conditions are checked to see whether sending a segment is OK. Note that this only occurs when there is data to send (i.e. not a window update or ack). The conditions to be checked are: a. Are sending a maximum sized segment. b. Have been idle and are depleting the output buffer. c. Are forcing output. d. Are sending more than half the maximum segment sent. e. Are retransmitting.
3.1.4. Window Update	Check to see whether sending a pure window update. Do so if the advertised window has changed by at least two maximum segments. Note that in this simulation, some of this code will never be executed; i.e. it should _always_ escape to indicate sending. The reason it is left is to preserve the logical structure and to allow for future flexibility.
3.1.5. Flags	If explicitly sending an acknowledgment, then make sure that the segment is sent.
3.1.6. Check Persist	Here, look at whether or not are in the persist state; which occurs if the buffer size is greater than zero, and have failed all the previous output conditions. So, the persist timer is also setup.
3.2. Send Output	The second half of output processing is to actually construct and send a segment, then to update state variables in the TCB. This is done in three steps, first the segment is constructed, then it is sent, and finally the various sequence numbers and the such like are updated.

3.2.1. Construct Output Message	Construct the output TCP segment by filling in all the appropriate fields; this includes length, sequence number, flags, windows and timestamps.
3.2.2. Send Output Message	The sending is done here, which is to queue up the segment. All queued messages are released to the BONeS environment upon completion of the TCP module.
3.2.3. Update Sequence Numbers	Having just sent the message, must update the various sequence numbers such as the maximum sequence number sent, and that sent but not yet acknowledged. What is done here is a first check to see whether output is because are not forced or retransmitting, and then first update the maximum and next sequence numbers, setting up an RTT timer (i.e. the RTT timer only occurs if sending new data, not retransmitting). Make sure setup for another retransmit too, if currently retransmitting that is.
3.3. First Init	Output processing will iterate if there are a number of segments to send. So, at the start do some initialising to set up a few things. Set up the forced output flag, the idle flag, and if idle then reset the congestion window.
3.4. Loop Init	Initialise information for each iteration of trying to send output, including resetting the iterator flag, and setting up the window offset, window size and ack flag.

4. Timer_Process

Kick in here on 100ms timer expiries that are generated from BONeS. Thump these down into 200ms or 500ms expires to correspond with TCP's fast and slow timers, respectively. The timer handlers ("Timer Fast Process", and "Timer Slow Processing") are then called if appropriate.

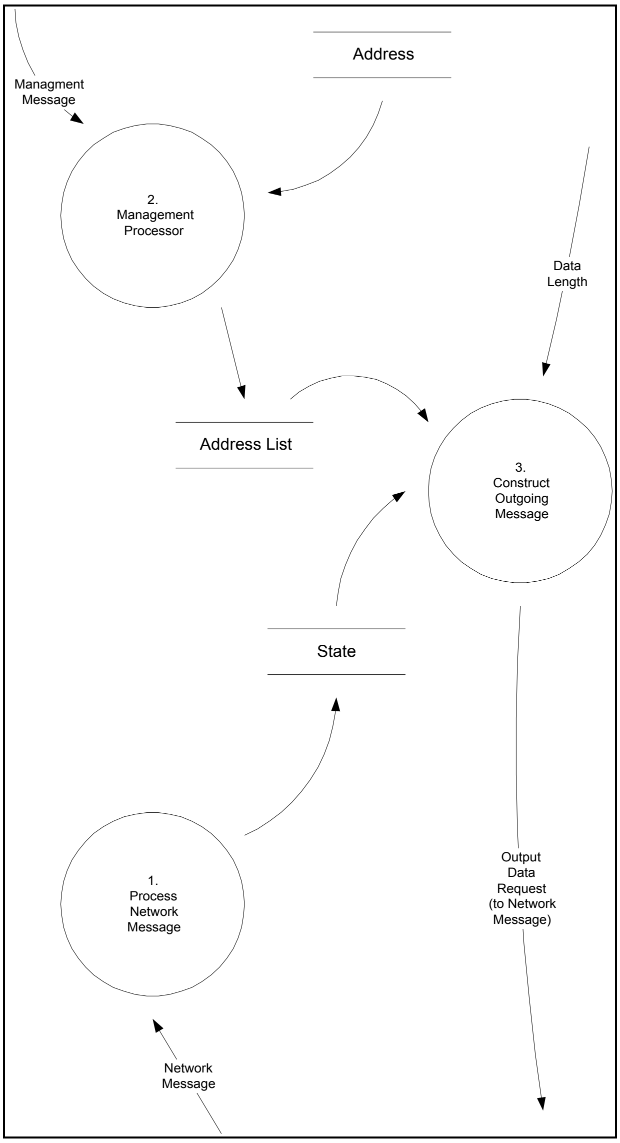
Function Name	*Description*
4.1. Timer Fast Process	The fast timer is used to schedule delayed acks; so check to see whether there is a delayed ACK pending, and if so, then go and pump it out via the output processing stage.
4.2. Timer Slow Process	The slow timer is used to schedule retransmits and persists, so check to see whether either of these timers have expired and if so, then go off and handle them. Also, ensure that updates to our idle counter (which is reset in input processing) and the RTT if timing a segment. Also increase TCP's virtual clock.

4.2.1. Retransmit Process	When a retransmit timer expires, first update out backoff value, schedule another timer event and fix up the congestion state. After which call output processing to start pumping data back into the pipe.
4.2.1.1. Update Backoff	Compute a new backoff value.
4.2.1.2. Setup Retransmit Timer	Schedule another retransmit timer by computing the time according to our RTT. Also reset the send sequence to be the start of our unacknowledged data, and reset the round trip time because it is not valid any more.
4.2.1.3. Update Congestion Information	Scale down the congestion window, because have lost data that was in the pipe. Also, reset duplicate ACKs count.
4.2.2. Persist Process	Process the persist timer, this means setup another persist timer and kick output processing with an indication to force output.
4.2.2.1. Setup Persist Timer	Setup the persist timer, do this by looking at the RTT mean and its variance, and our computed backoff value. The persist timer is then scheduled and the backoff increases for the next persist (should it come around).

2.4. Network-Adaption Layer

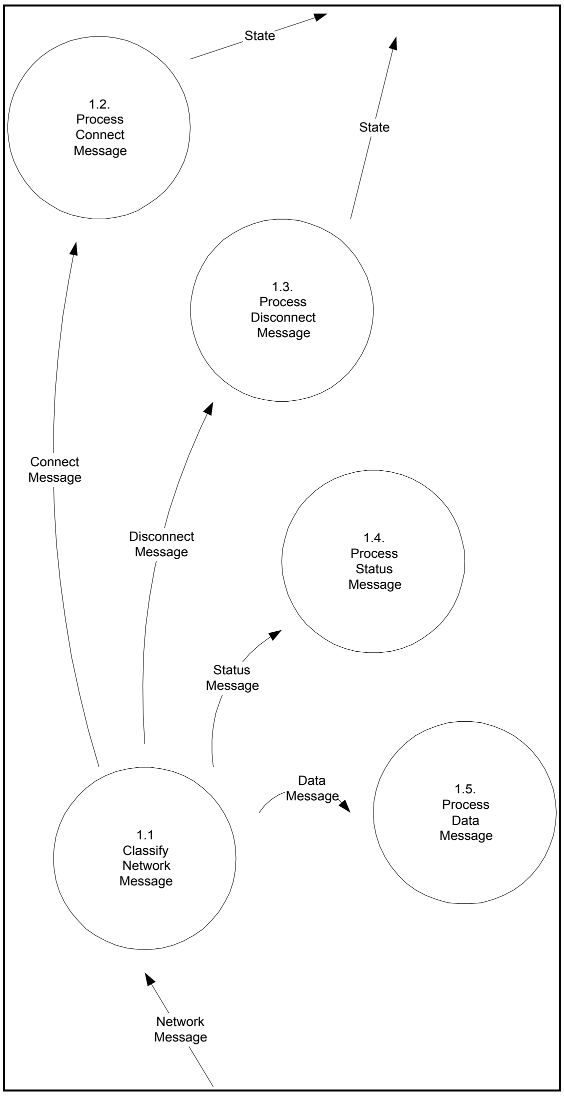
DFD 0: Top

The Top Level DFD delineates the major processing blocks, showing the data relationships between them. The *State* refers to the currently known state of the Network Layer, and *Address List* representing the abstraction described above with *Address* being the mandatory item for Management.


Address
Managment Message
2. Management Processor
Data Length
Address List
3. Construct Outgoing Message
State
Output Data Request (to Network Message)
1. Process Network Message
Network Message

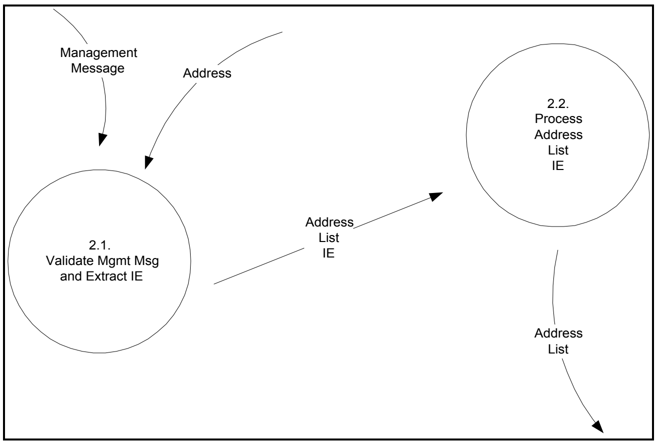
DFD 1: Process Network Message

The *Process Network Message* is responsible for interpreting and acting upon messages arriving from the Network Layer. Only the *Network Connect Indication Message* and *Network Disconnect Indication Message* have any effect here, and are used to update the known *State* of the Network Layer. A separate process is defined for each message.



DFD 2: Management Processor

The *Management Processor* has a straight forward partitioning. Firstly, the message is validated in *Validate Mgmt Message and Extract IE* to ensure that it has the correct destination address and content, after which the content is processed according its type. The only specific content processed at this point in time is the *Network-Adaption Address List IE* in *Process Address List IE* -- this results in a change to the *Address List*


Management Message
Address
2.2. Process Address List IE
2.1. Validate Mgmt Msg and Extract IE
Address List IE
Address List

PSPEC 3: Construct Outgoing Message

The given *Data Length* is used to construct a *Network Data Request Message* using an *Address* randomly selected from the *Address List*. However, this will only occur if the *State* of the Network Layer is *True*. Note also that the *Network Data Request Message* does have a content, but it is an *Application Data Message* so that if any intermediate entity decides to interrogate the message, they will find a content that represents an abstract unit of data only.

```
Inputs:
    Data_Length: Integer
    State: Boolean
    Address_List: ARRAY OF Integer
Outputs:
    Output_Data_Req: Network Data Request Message
Operation:
    1. DECLARE Address: Integer
    2. DELCARE Number: Integer
    3. DECLARE Data_Msg: Application Data Message
    4. IF State = True THEN
        1. Number := RANDOM_UNIFORM (0, Length (Address_List))
        2. Address := Address_List [Number]
        3. Data_Msg := ConstructMsg_Applic_Data (Data_Length)
        4. Output_Data_Req :=
               ConstructMsg_N_Data_Req (Address, Data_Msg)
    5. STOP
```

PSPEC 1.1: Classify Network Message

The message must be classified according to its type so that it can be processed by the appropriate task. This is done by looking at the type of the message.

```
Inputs:
    Network_Msg: Message
Outputs:
    Connect_Msg: Network Connect Indication Message
    Disconnect_Msg: Network Disconnect Indication Message
    Status_Msg: Network Status Indication Message
    Data_Msg: Network Data Indication Message
Operation:
    1. If Type (Connect_Msg) = Type (Network_Msg) THEN
        1. Connect_Msg := Network_Msg
    2. If Type (Disconnect_Msg) = Type (Network_Msg) THEN
        1. Disconnect_Msg := Network_Msg
    3. If Type (Status_Msg) = Type (Network_Msg) THEN
        1. Status_Msg := Network_Msg
    4. If Type (Data_Msg) = Type (Network_Msg) THEN
        1. Data_Msg := Network_Msg
    5. STOP
```

PSPEC 1.2: Process Connect Message

The message is used to indicate the current state of the Network Layer.

```
Inputs:
    Connect_Msg: Network Connect Indication Message
Outputs:
    State: Boolean
Processing:
    1. State := True
    2. STOP
```

PSPEC 1.3: Process Disconnect Message

The message is used to indicate the current state of the Network Layer.

```
Inputs:
    Disconnect_Msg: Network Disconnect Indication Message
Outputs:
    State: Boolean
Processing:
    1. State := False
    2. STOP
```

PSPEC 1.4: Process Status Message

The message is currently not processed.

```
Inputs:
    Status_Msg: Network Status Indication Message
Outputs:
    n/a
Processing:
    1. STOP
```

PSPEC 1.5: Process Data Message

The message is currently not processed.

```
Inputs:
    Data_Msg: Network Data Indication Message
Outputs:
    n/a
Processing:
    1. STOP
```

PSPEC 2.1: Validate Mgmt Message and Extract IE

The *Management Message* is inspected to ensure that is destined for this module, and that the content is valid via *Validate Mgmt Message and Extract IE*. The appropriate output is generated depending on the type of content.

```
Inputs:
    Mgmt_Msg: Management Set Indication Message
    Address: Integer
Outputs:
    Address_List_IE: Network-Adaption Address List IE
Processing:
    1. DECLARE Unknown_IE: IE
    2. IF Address (Mgmt_Msg) = Address THEN
        1. Unknown_IE := ExtractMsg_M_Set_Ind (Mgmt_Msg)
        2. IF Type (Unknown_IE) = Type (Address_List_IE)
            1. Address_List_IE := Unknown_IE
    3. STOP
```

PSPEC 2.2: Process Address List IE

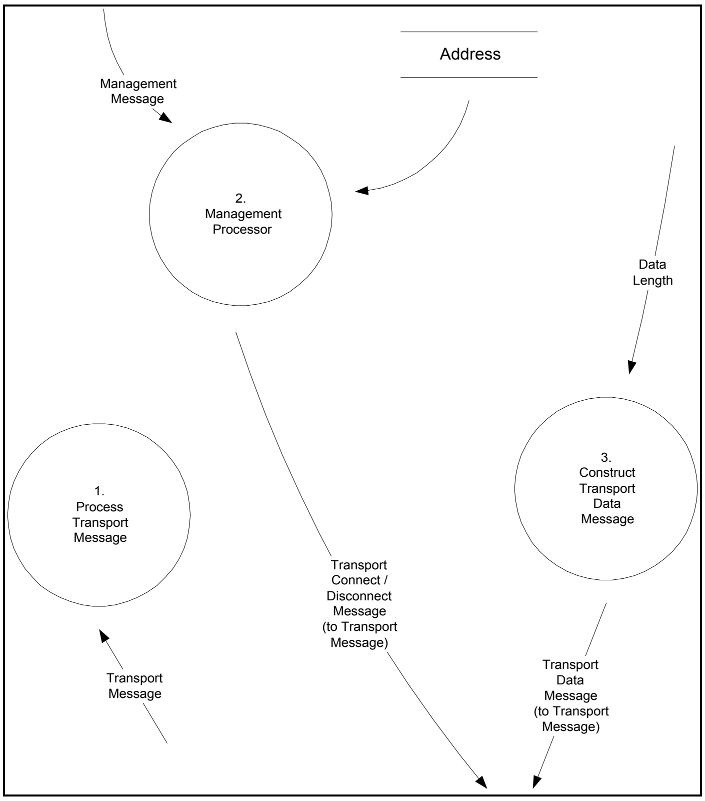
The content of the *Address List IE* is processed in *Process Address List IE* and used to update the internal *Address List*.

```
Inputs:
    Address_List_IE: Network-Adaption Address List IE
Outputs:
    Address_List : ARRAY OF Integer
Processing:
    1. Address_List := ExtractIE_NA_Address_List (Address_List_IE)
    2. STOP
```

2.5. Transport-Adaption Layer

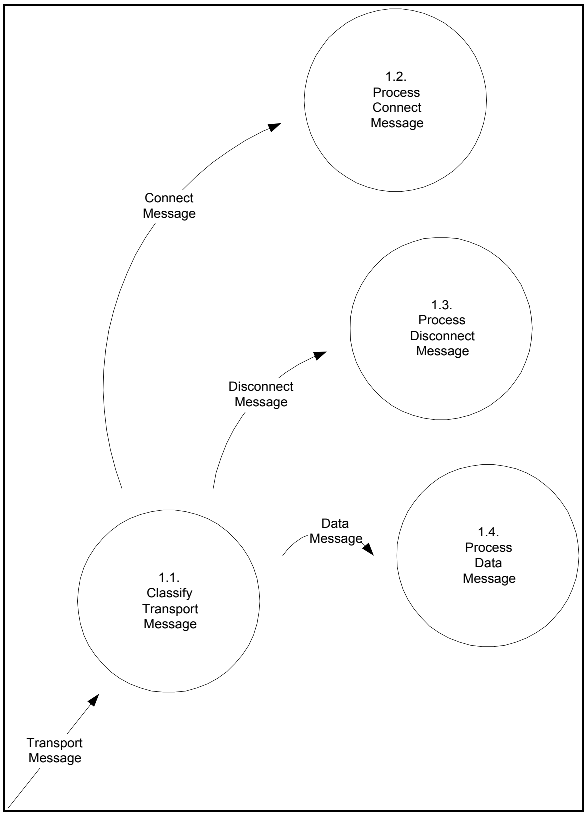
DFD 0: Top

The architectural delineation can be seen in the Top Level DFD. The only data store here is the *Address*, being the mandatory item for Management.


Address
Management Message
2. Management Processor
Data Length
1. Process Transport Message
3. Construct Transport Data Message
Transport Connect / Disconnect Message (to Transport Message)
Transport Message
Transport Data Message (to Transport Message)

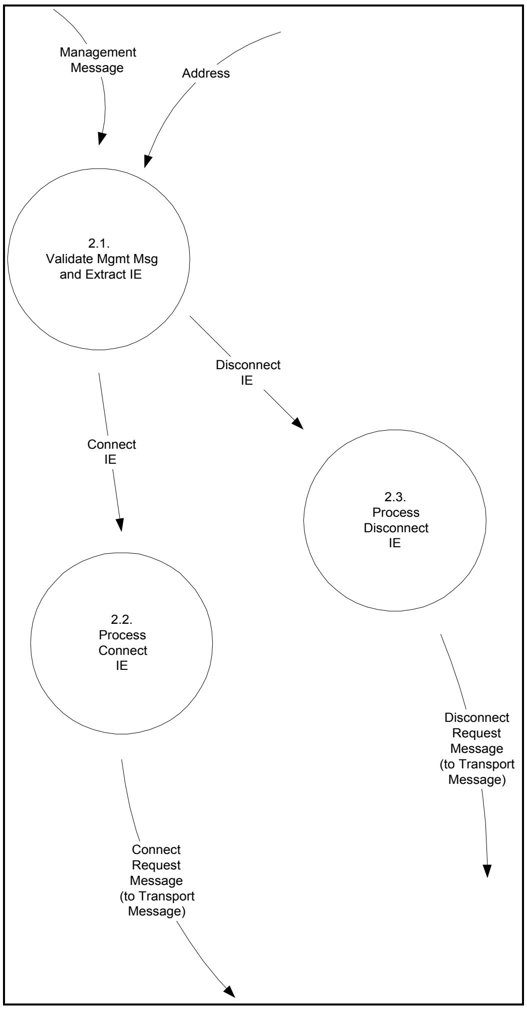
DFD 1: Process Transport Message

The *Process Transport Message* is responsible for interpreting and acting upon messages arriving from the Transport Layer. Processes are provided *for Transport Connect Indication*, *Transport Disconnect Indication* and *Transport Data Indication* Messages.



DFD 2: Management Processor

The *Management Processor* has a straight forward partitioning. Firstly, the message is validated in *Validate Mgmt Message and Extract IE* to ensure that it has the correct destination address and content, after which the content is processed according its type. There are two specific IEs that are acted upon, that being the *Transport-Adaption Connect IE* and the *Transport-Adaption Disconnect IE*. These result in the generation of *Transport Connect Request* and *Transport Disconnect Request* Messages, respectively.


Management Message
Address
2.1. Validate Mgmt Msg and Extract IE
Disconnect IE
Connect IE
2.3. Process Disconnect IE
2.2. Process Connect IE
Disconnect Request Message (to Transport Message)
Connect Request Message (to Transport Message)

PSPEC 3: Construct Outgoing Message

The given *Data Length* is used to construct a *Transport Data Request Message*. Note that the *Transport Data Request Message* does have a content, but it is an *Application Data Message* so that if any intermediate entity decides to interrogate the message, they will find a content that represents an abstract unit of data only.

```
Inputs:
    Data_Length: Integer
Outputs:
    Output_Data_Req: Transport Data Request Message
Operation:
    1. DECLARE Data_Msg: Application Data Message
    2. Data_Msg := ConstructMsg_Applic_Data (Data_Length)
    3. Output_Data_Req := ConstructMsg_T_Data_Req (Data_Msg)
    4. STOP
```

PSPEC 1.1: Classify Transport Message

The message must be classified according to its type so that it can be processed by the appropriate task. This is done by looking at the type of the message.

```
Inputs:
    Transport_Msg: Message
Outputs:
    Connect_Msg: Transport Connect Indication Message
    Disconnect_Msg: Transport Disconnect Indication Message
    Data_Msg: Transport Data Indication Message
Operation:
    1. If Type (Connect_Msg) = Type (Transport_Msg) THEN
        1. Connect_Msg := Transport_Msg
    2. If Type (Disconnect_Msg) = Type (Transport_Msg) THEN
        1. Disconnect_Msg := Transport_Msg
    3. If Type (Data_Msg) = Type (Transport_Msg) THEN
        1. Data_Msg := Transport_Msg
    4. STOP
```

PSPEC 1.2: Process Connect Message

The message is currently not processed.

```
Inputs:
    Connect_Msg: Transport Connect Indication Message
Outputs:
    n/a
Processing:
    1. STOP
```

PSPEC 1.3: Process Disconnect Message

The message is currently not processed.

```
Inputs:
    Disconnect_Msg: Transport Disconnect Indication Message
Outputs:
    n/a
Processing:
    1. STOP
```

PSPEC 1.4: Process Data Message

The message is currently not processed.

```
Inputs:
    Data_Msg: Transport Data Indication Message
Outputs:
    n/a
Processing:
    1. STOP
```

PSPEC 2.1: Validate Mgmt Message and Extract IE

The *Management Message* is inspected to ensure that is destined for this module, and that the content is valid via *Validate Mgmt Message and Extract IE*. The appropriate output is generated depending on the type of content.

```
Inputs:
    Mgmt_Msg: Management Set Indication Message
    Address: Integer
Outputs:
    Connect_IE: Transport-Adaption Connect IE
    Disconnect_IE: Transport-Adaption Disconnect IE
Processing:
    1. DECLARE Unknown_IE: IE
    2. IF Address (Mgmt_Msg) = Address THEN
        1. Unknown_IE := ExtractMsg_M_Set_Ind (Mgmt_Msg)
        2. IF Type (Unknown_IE) = Type (Connect_IE)
            1. Connect_IE := Unknown_IE
        3. IF Type (Unknown_IE) = Type (Disconnect_IE)
            1. Disconnect_IE := Unknown_IE
    3. STOP
```

PSPEC 2.2: Process Connect IE

The content of the *Connect IE* is processed in *Process Connect IE*, which generates a *Transport Connect Request Message*.

```
Inputs:
    Connect_IE: Transport-Adaption Connect IE
Outputs:
    Connect_Msg: Transport Connect Request Message
Processing:
    1. DECLARE Address: Integer
    2. Address := ExtractIE_TA_Connect (Connect_IE)
    3. Connect_Msg := ConstructMsg_T_Connect_Req (Address)
    4. STOP
```

PSPEC 2.2: Process Disconnect IE

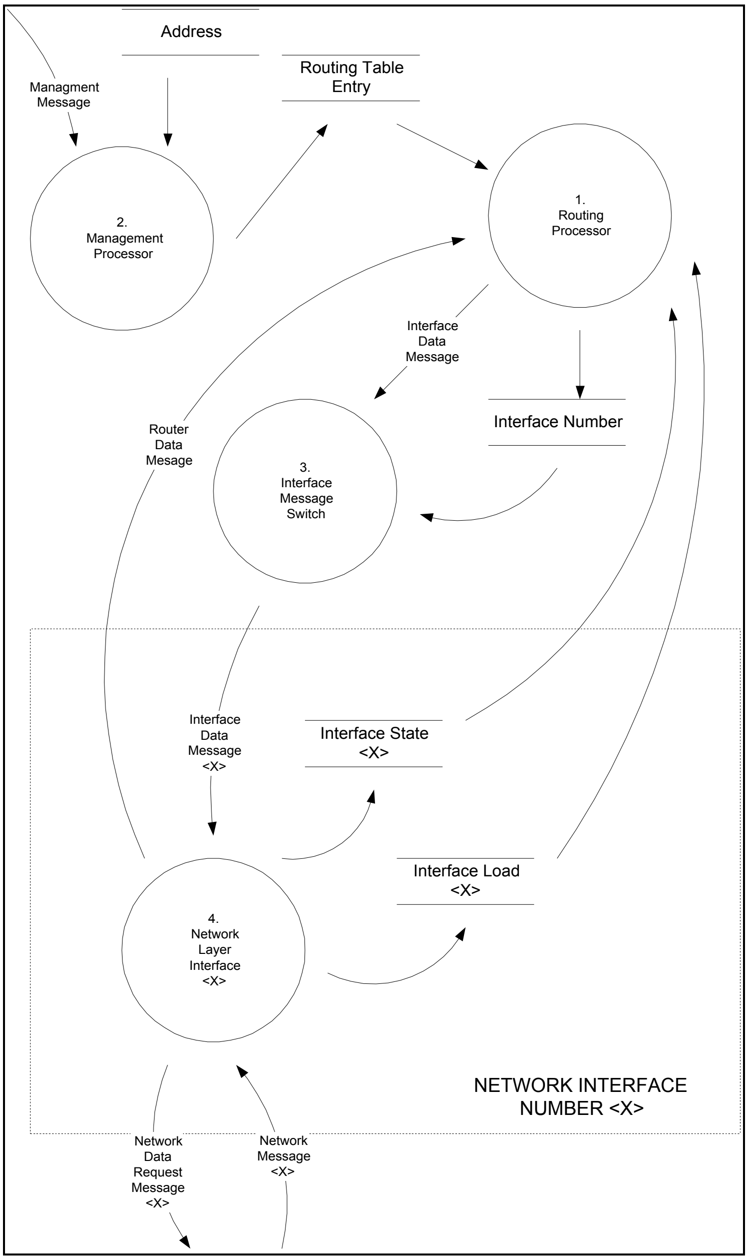
The content of the *Disconnect IE* is processed in *Process Disconnect IE*, which generates a *Transport Disconnect Request Message*.

```
Inputs:
    Disconnect_IE: Transport-Adaption Disconnect IE
Outputs:
    Disconnect_Msg: Transport Disconnect Request Message
Processing:
    1. Disconnect_Msg := ConstructMsg_T_Disconnect_Req ()
    2. STOP
```

2.6. Routing-Module

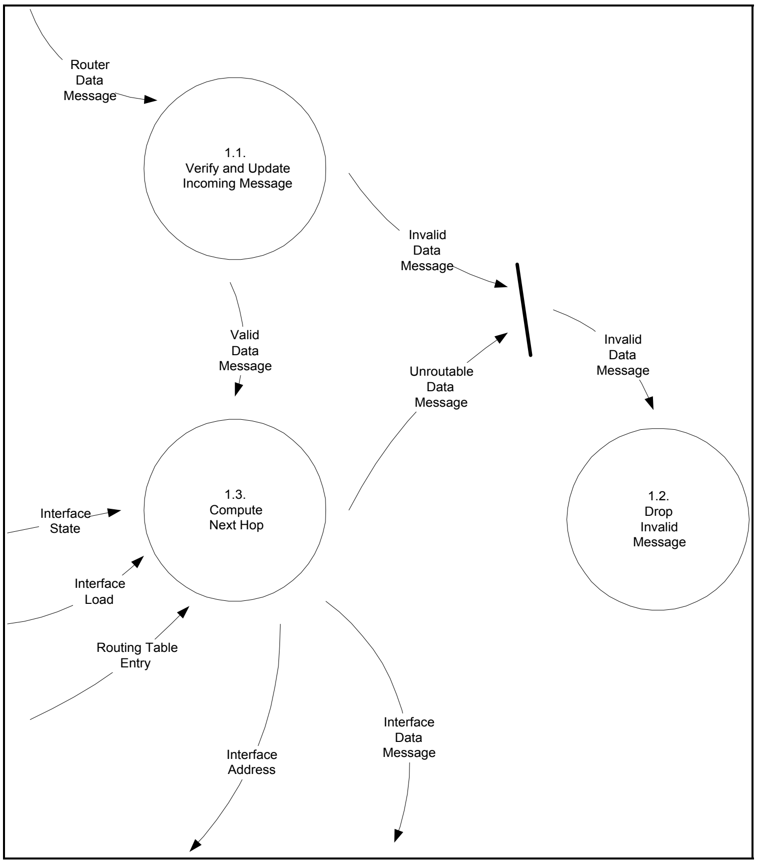
DFD 0: Top

In the Top Level DFD, the *Management Processor* uses the module's *Address* and updates *Routing Table Entries*. These are read by the *Routing Processor*, which uses the *Interface State* and *Interface Load* updated by each *Network Layer Interface*. The *Interface Message Switch* will switch a message to a specific interface. The *"<X>"* qualification indicates that there are multiple instances item; and a process must supply the qualifier for the specific instance.



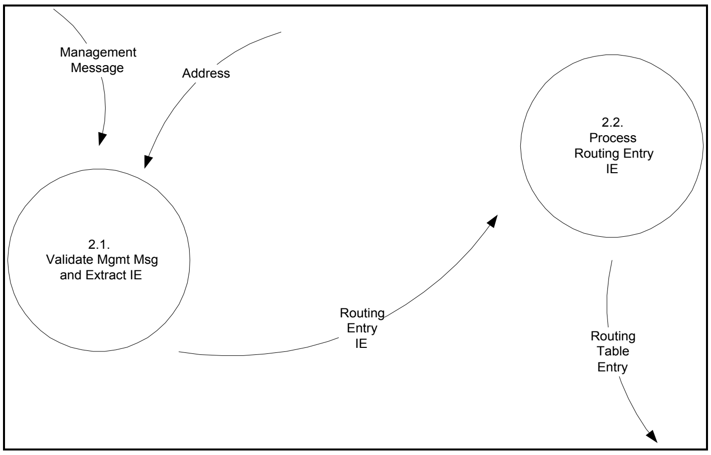
DFD 1: Routing Module

The *Routing Module* is responsible for locating an interface for the given message. It must first *Verify and Update [the] Incoming Message* taking care of Hop Count (Time-To-Live) fields; the message is dropped using *Drop Invalid Message* if this verification fails. Having had this occur, the next *Interface Address* for the *Valid Data Message* is located in *Compute Next Hop*. This computation involves interface specific state and general routing information. If no route can be found, then the *Unroutable Data Message* is dropped; otherwise it is passed out as an *Interface Data Message* to be processed by a specific *Network Interface*.



DFD 2. Management Processor

The *Management Processor* has a straight forward partitioning. Firstly, the message is processed in *Validate Mgmt Message and Extract IE* to ensure that it has the correct destination address and content, after which the content is processed according its type. The only specific content processed at this point in time is the *Routing-Module Routing Entry IE* in *Process Routing Entry IE* -- this results in a change to a *Routing Table Entry*


Management Message
Address
2.1. Validate Mgmt Msg and Extract IE
2.2. Process Routing Entry IE
Routing Entry IE
Routing Table Entry

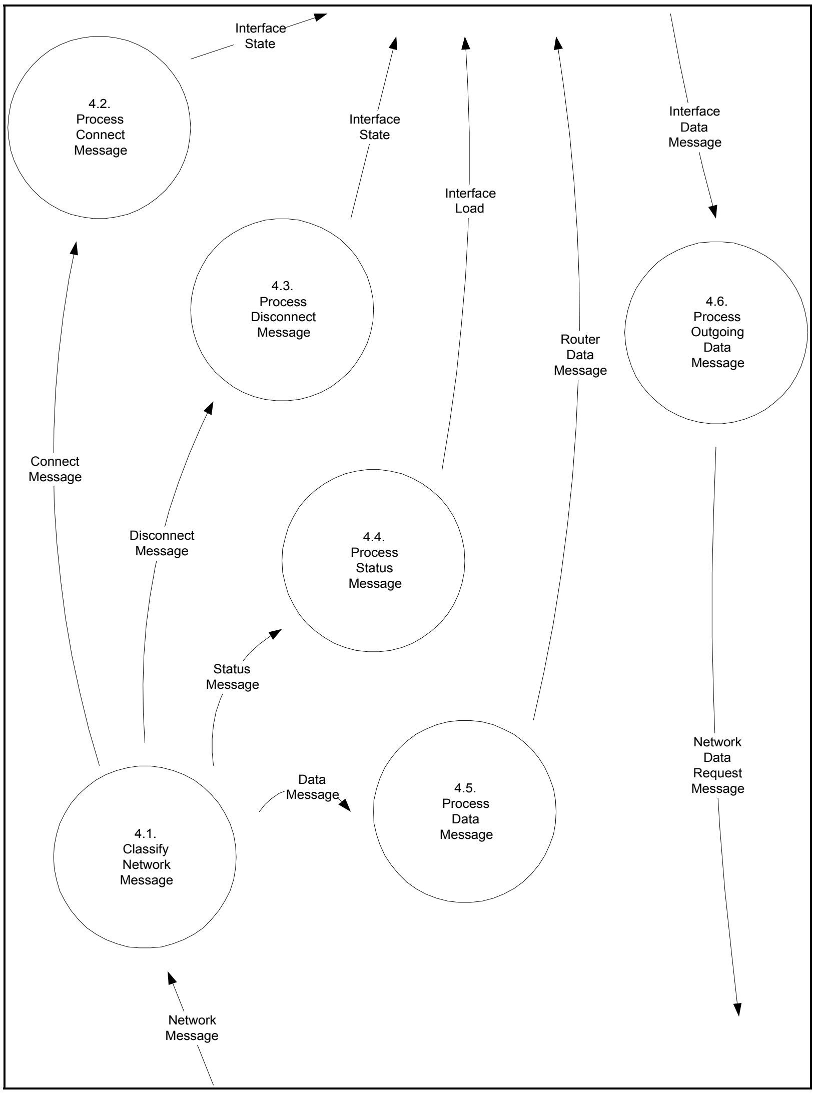
PSPEC 3. Interface Message Switch

The given *Interface Data Message* is switched to a specific output depending upon the given *Interface Number*.

```
Inputs:
    Interface_Data_Msg: Network Data Indication Message
    Interface_Number: Integer
Outputs:
    Interface_Data_Msg<X> : Network Data Indication Message
Operation:
    1. Interface_Data_Msg<Interface_Number> := Inteface_Data_Msg
    2. STOP
```

DFD 4. Network Layer Interface

The *Network Layer Interface* is architecturally divided into processing messages arriving from the Network Layer, and processing messages destined for the Network Layer. Messages from, are first classified in *Classify Network Message* and thence switched to be processed by a specific task. *Process Connect Message* and *Process Disconnect Message* results in a change to the *Interface State,* whereas *Process Status Message* results in a change to the *Interface Load* and *Process Data Message* passes the message as a *Router Data Indication* for routing. Outgoing messages, *Interface Data Messages*, are transformed into a *Network Data Request Message* for output.


Interface State
4.2. Process Connect Message
Interface State
Interface Data Message
Interface Load
4.3. Process Disconnect Message
Router Data Message
4.6. Process Outgoing Data Message
Connect Message
Disconnect Message
4.4. Process Status Message
Status Message
Network Data Request Message
Data Message
4.5. Process Data Message
4.1. Classify Network Message
Network Message

PSPEC 1.1. Verify and Update Incoming Message

In *Verify and Update Incoming Message*, the *Router Data Message* has its hop count field decremented and then checked to see whether or not it is zero: if it is, then the message is considered to be an *Invalid Data Message*, otherwise the modified message is output as a *Valid Data Message*.

```
Inputs:
    Router_Data_Msg: Network Data Indication Message
Outputs:
    Invalid_Data_Msg: Network Data Indication Message
    Valid_Data_Msg: Network Data Indication Message
Operation:
    1. DECLARE Hop_Count: Integer
    2. Hop_Count := ExtractMsg_N_Data_Ind (Router_Data_Msg,HOPCOUNT)
    3. Hop_Count := Hop_Count - 1
    4. InsertMsg_N_Data_Ind (Router_Data_Msg, HOPCOUNT, Hop_Count)
    4. IF Hop_Count = 0 THEN
        1. Invalid_Data_Msg := Router_Data_Msg
    5. IF Hop_Count > 0 THEN
        1. Valid_Data_Msg := Router_Data_Msg
    6. STOP
```

PSPEC 1.2. Drop Invalid Message

In *Drop Invalid Message,* the *Invalid Data Message* is dropped after a suitable log is made.

```
Inputs:
    Invalid_Data_Msg: Network Data Indication Message
Outputs:
    n/a
Operation:
    1. LOG ("Dropping Message due to Problem:")
    2. LOG (Invalid_Data_Msg)
    3. STOP
```

PSPEC 1.3. Compute Next Hop

In *Compute Next Hop*, an *Interface Address* must be determined for the *Valid Data Message*. This is done by looking through all the *Routing Table Entries* for the message's *Address* in order to select an Interface with the least cost. This function is the critical centre of the router, and where most time will be spent.

```
Inputs:
    Valid_Data_Msg: Network Data Indication Message
    Interface_State: ARRAY OF Boolean
    Interface_Load: ARRAY OF Real
    Routing_Table_Entry: MATRIX OF Real
Outputs:
    Interface_Address: Integer
    Interface_Data_Msg: Network Data Indication Message
    Unroutable_Data_Msg: Network Data Indication Message
Operation:
    1. DECLARE InterfaceCost: Real
    2. DECLARE InterfaceNumber: Integer
    3. DECLARE InterfaceCount: Integer
    4. DECLARE Address: Integer
    5. Address := ExtractMsg_N_Data_Ind (Valid_Data_Msg, ADDRESS)
    6. InterfaceNumber := -1
    7. InterfaceCount := 0
    Label_Loop_Next:
    8. IF Interface_State[InterfaceCount] = True THEN
        1. DECLARE Cost: Real
        2. Cost := ComputeCost (Address, InterfaceCount,
                           Interface_Load, Routing_Table_Entry)
        3. IF Cost > InterfaceCost THEN
            1. InterfaceCost := Cost
            2. InterfaceNumber := InterfaceCount
    9. InterfaceCount := InterfaceCount + 1
    10. IF InterfaceCount < MAXIMUM_INTERFACES THEN Label_Loop_Next
    11. IF InterfaceCount = -1 THEN
        1. Unroutable_Data_Msg := Valid_Data_Msg
    12. IF InterfaceCount != -1 THEN
        1. Interface_Address := InterfaceNumber
        2. Interface_Data_Msg := Valid_Data_Msg
    13. STOP
```

FUNCTION 1.3.1. ComputeCost

This PSPEC uses a function that computes a "weighted cost" by using the interfaces defined cost, and weighing it according to the current load on that interface. This has been encapsulated within a function so as to allow modifications. Note that BETA is the defined weighing factor, whose value is not specifically known and can be arbitrary set.

```
Function:
    ComputeCost
Inputs:
    Address: Integer
    Interface: Integer
    Interface_Load: Real
    Routing_Table_Entry: MATRIX OF Real
Outputs:
    Cost: Real
Operation:
    1. Cost := Routing_Table_Entry[Address][Interface] +
                   Interface_Load[Interface] * BETA
    2. STOP
```

PSPEC 2.1. Validate Mgmt Message and Extract IE

The *Management Message* is inspected to ensure that is destined for this module, and that its content is valid using *Validate Mgmt Message and Extract IE*. The appropriate output is generated depending on the type of content.

```
Inputs:
    Mgmt_Msg: Management Set Indication Message
    Address: Integer
Outputs:
    Routing_Entry_IE: Routing-Module Routing Entry IE
Processing:
    1. DECLARE Unknown_IE: IE
    2. IF Address (Mgmt_Msg) = Address THEN
        1. Unknown_IE := ExtractMsg_M_Set_Ind (Mgmt_Msg)
        2. IF Type (Unknown_IE) = Type (Routing_Entry_IE)
            1. Routing_Entry_IE := Unknown_IE
    3. STOP
```

PSPEC 2.2. Process Routing Entry IE

The content of the *Routing Entry IE* is processed in *Process Routing Entry IE* and used to update a *Routing Table Entry*. Note that the *Routing Table Entry* is a matrix keyed by the *Address* and *Interface* relating to the Route. The *Cost* is the component stored in the matrix.

```
Inputs:
    Routing_Entry_IE: Routing-Module Routing Entry IE
Outputs:
    Routing_Table_Entry: MATRIX OF Real
Processing:
    1. DECLARE Address: Integer
    2. DECLARE Interface: Integer
    3. DECLARE Cost: Real
    4. Address :=
           ExtractIE_RM_Routing_Entry (Routing_Entry_IE, ADDRESS)
    5. Interface :=
           ExtractIE_RM_Routing_Entry (Routing_Entry_IE, INTERFACE)
    6. Cost :=  ExtractIE_RM_Routing_Entry (Routing_Entry_IE, COST)
    5. Routing_Table_Entry[Address][Interface] := Cost
    6. STOP
```

PSPEC 4.1. Classify Network Message

The message must be classified according to its type so that it can be processed by the appropriate task. This is done by looking at the type of the message.

```
Inputs:
    Network_Msg: Message
Outputs:
    Connect_Msg: Network Connect Indication Message
    Disconnect_Msg: Network Disconnect Indication Message
    Status_Msg: Network Status Indication Message
    Data_Msg: Network Data Indication Message
Operation:
    1. If Type (Connect_Msg) = Type (Network_Msg) THEN
        1. Connect_Msg := Network_Msg
    2. If Type (Disconnect_Msg) = Type (Network_Msg) THEN
        1. Disconnect_Msg := Network_Msg
    3. If Type (Status_Msg) = Type (Network_Msg) THEN
        1. Status_Msg := Network_Msg
    4. If Type (Data_Msg) = Type (Network_Msg) THEN
        1. Data_Msg := Network_Msg
    5. STOP
```

PSPEC 4.2. Process Connect Message

The message is used to indicate the current state of the Network Layer. Note that this *Interface State* is particular to our interface, i.e. it is parameterised at an upper level.

```
Inputs:
    Connect_Msg: Network Connect Indication Message
Outputs:
    Interface_State: Boolean
Processing:
    1. Interface_State := True
    2. STOP
```

PSPEC 4.3. Process Disconnect Message

The message is used to indicate the current state of the Network Layer. Note that this Interface State is particular to our interface, i.e. it is parameterised at an upper level.

```
Inputs:
    Disconnect_Msg: Network Disconnect Indication Message
Outputs:
    Interface_State: Boolean
Processing:
    1. Interface_State := False
    2. STOP
```

PSPEC 4.4. Process Status Message

The message is used to indicate the current load state of the Network Layer; this is used in the determination of a next hop. Note that this *Interface Load* is particular to our interface, i.e. it is parameterised at a global level.

```
Inputs:
    Status_Msg: Network Status Indication Message
Outputs:
    Interface_Load: Real
Processing:
    1. DECLARE Load_IE: Network Load IE
    2. Load_IE := ExtractMsg_N_Status_Ind (Status_Msg, IE)
    3. Interface_Load := ExtractIE_N_Load (Load_IE)
    4. STOP
```

PSPEC 4.5. Process Data Message

The message is passed through, unaltered.

```
Inputs:
    Data_Msg: Network Data Indication Message
Outputs:
    Router_Data_Msg: Network Data Indication Message
Processing:
    1. Router_Data_Msg := Data_Msg
    2. STOP
```

PSPEC 4.6. Process Outgoing Data Message

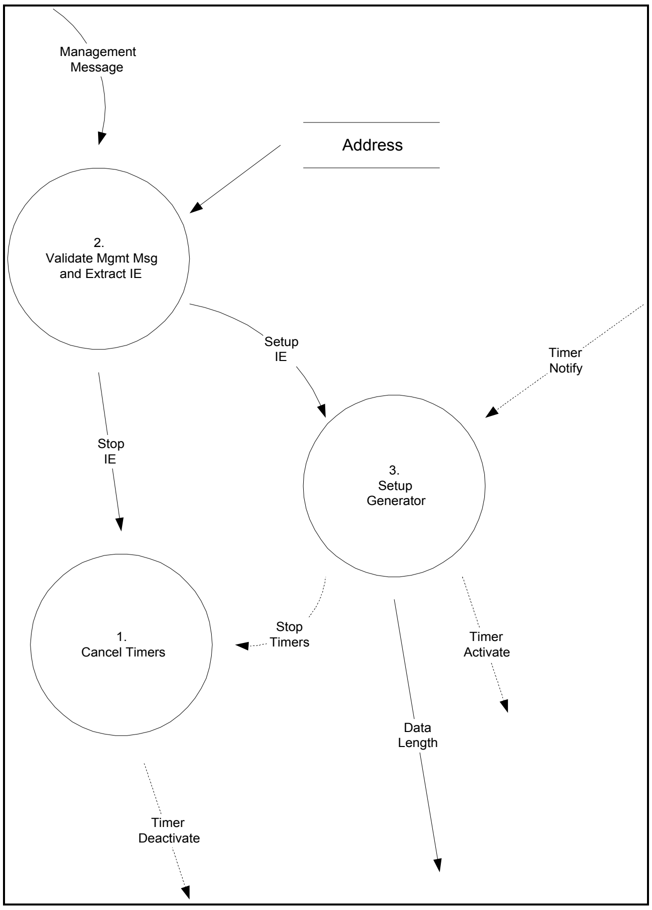
The message is converted from an indication to a request so that it can be passed to the Network Layer.

```
Inputs:
    Interface_Data_Msg: Network Data Indication Message
Outputs:
    Network_Data_Request_Msg: Network Data Request Message
Processing:
    1. Network_Data_Request_Msg :=
            ConvertMsg_N_Data_Ind_To_Req (Interface_Data_Msg)
    2. STOP
```

2.7. Generator

DFD 0: Top

The Top Level DFD illustrates the elements that receive and process messages from Management. Firstly, the message is validated in *Validate Mgmt Message and Extract IE* to ensure that it has the correct destination *Address* and content, after which the content is processed according its type. A *Setup IE* is dealt with by the *Setup Generator* process, and the *Stop IE* by the *Cancel Timers* process. The *Setup Generator* itself can also initiate a *Cancel Timers*.



PSPEC 1: Cancel Timers

The current timer is cancelled, to prevent further scheduling of generator activity.

```
Inputs:
    Stop_IE : Generator Stop IE
    Stop_Timers: SIGNAL
Outputs:
    Timer_Deactivate: SIGNAL
Operation:
    1. Timer_Deactivate := STOP_TIMER ()
    2. STOP
```

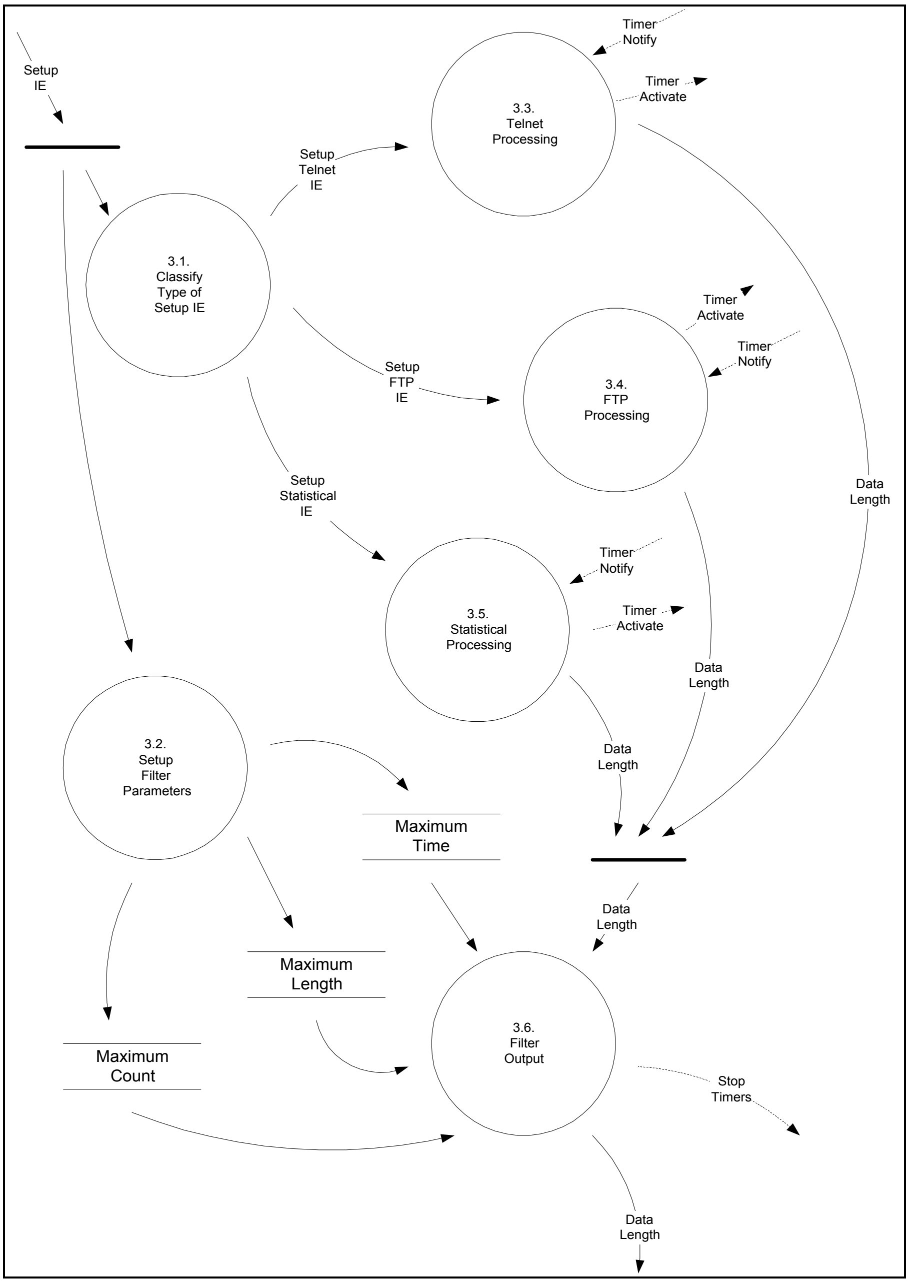
PSPEC 2: Validate Mgmt Message and Extract IE

The *Management Message* is inspected to ensure that is destined for this module, and that the content is valid, via *Validate Mgmt Message and Extract IE*. The appropriate output is generated depending on the type of content.

```
Inputs:
    Mgmt_Msg: Management Set Indication Message
    Address: Integer
Outputs:
    Setup_IE: Generator Setup IE
    Stop_IE: Generator Stop IE
Processing:
    1. DECLARE Unknown_IE: IE
    2. IF Address (Mgmt_Msg) = Address THEN
        1. Unknown_IE := ExtractMsg_Mgmt_Set_Ind (Mgmt_Msg)
        2. IF Type (Unknown_IE) = Type (Setup_IE)
            1. Setup_IE := Unknown_IE
        3. IF Type (Unknown_IE) = Type (Stop_IE)
            1. Stop_IE := Unknown_IE
    3. STOP
```

DFD 3: Setup Generator

The *Setup Generator* carries out three main tasks. The first task is the classification of the *Setup IE* and subsequent passing to the respective process that will generate output using the conveyed profile parameters. The second task is the extraction and setup of filter parameters in *Setup Filter Parameters*. These parameters are used in the third task of filtering output, in *Filter Output*. The latter will either allow *Data Length* to pass through, or signal the *Cancel Timers* to prevent any more scheduling if a limitation has been reached.



PSPEC 3.1: Classify Type of Setup IE

The input *Setup IE* is further classified according to the type of profile that it is setting up. It is then placed onto the corresponding output.

```
Inputs:
    Setup_IE: Generator Setup IE
Outputs:
    Setup_Telnet_IE: Generator Setup Telnet IE
    Setup_FTP_IE: Generator Setup FTP IE
    Setup_Statistical_IE: Generator Setup Statistical IE
Operation:
    1. IF Type (Setup_IE) = Type (Setup_Telnet_IE) THEN
        1. Setup_Telnet_IE := Setup_IE
    2. IF Type (Setup_IE) = Type (Setup_FTP_IE) THEN
        1. Setup_FTP_IE := Setup_IE
    3. IF Type (Setup_IE) = Type (Setup_Statistical_IE) THEN
        1. Setup_Statistical_IE := Setup_IE
    4. STOP
```

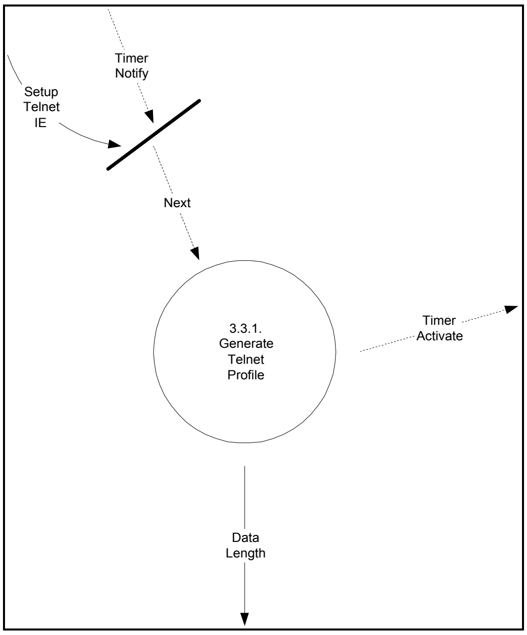
PSPEC 3.2: Setup Filter Parameters

The *Setup IE* also contains parameters indicating limitations that are to be imposed via. the output filter. *Setup Filter Parameters* is responsible for extracting these parameters and holding them.

```
Inputs:
    Setup_IE: Generator Setup IE
Outputs:
    Maximum_Time: Real
    Maximum_Length: Integer
    Maximum_Count: Integer
Operation:
    1. Maximum_Time := ExtractIE_G_Setup_IE (Setup_IE, MAX_TIME)
    2. Maximum_Length := ExtractIE_G_Setup_IE (Setup_IE, MAX_LENGTH)
    3. Maximum_Count := ExtractIE_G_Setup_IE (Setup_IE, MAX_COUNT)
    4. STOP
```

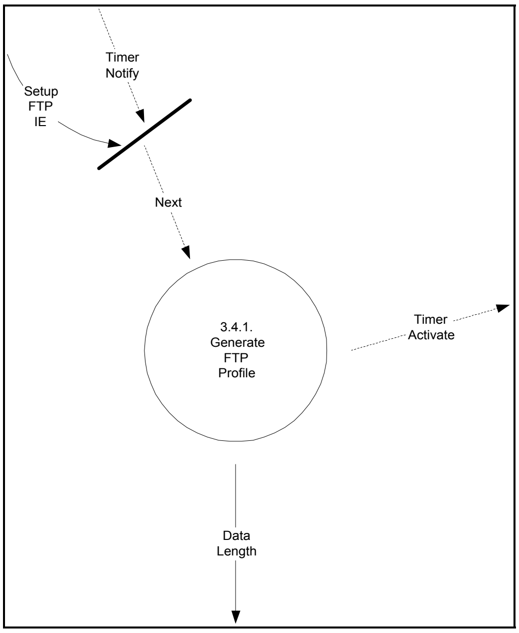
DFD 3.3: Telnet Processing

In *Telnet Processing* the arrival of either a *Timer Notify* due to expiry, or a *Setup Telnet IE* will trigger the generation of the next item using *Generate Telnet Profile*. This will set up a timer, by way of *Timer Activate*, and generate a *Data Length*.



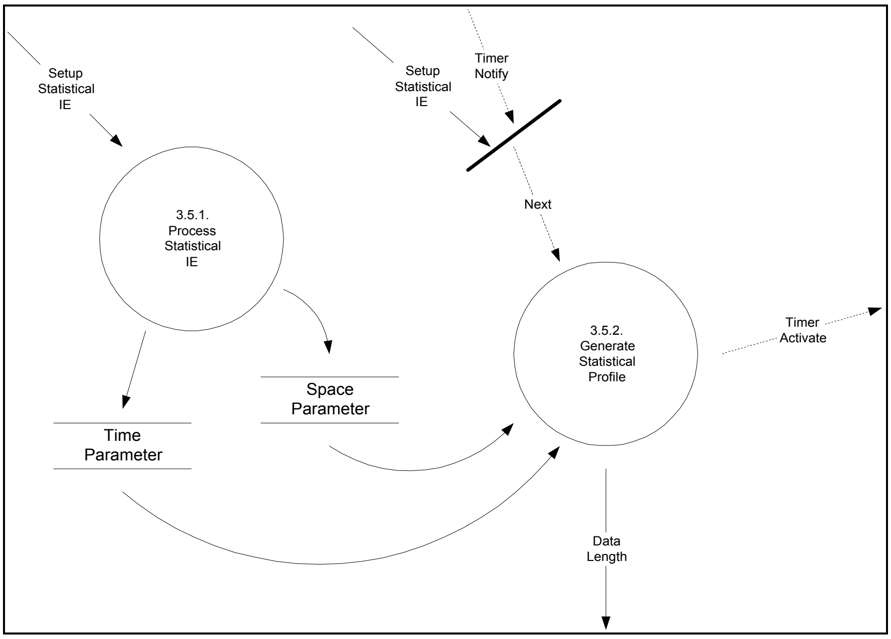
DFD 3.4: FTP Processing

In *FTP Processing* the arrival of either a *Timer Notify* due to expiry, or a *Setup FTP IE* will trigger the generation of the next item using *Generate FTP Profile*. This will set up a timer, by way of *Timer Activate*, and generate a *Data Length*.



DFD 3.5: Statistical Processing

In *Statistical Processing* the arrival of a *Setup Statistical IE* is processed by *Process Statistical IE* which extracts the *Timer Parameter* and *Space Parameter*. Also, both the *Process Statistical IE* and a *Timer Notify*, due to expiry, will trigger the generation of the next time using the two parameters. This will set up a timer, by way of *Timer Activate*, and generate a *Data Length*.



PSPEC 3.6: Filter Output

When *Data Length* is generated and placed for output, the *Filter Output* is used to ensure that the generation of this item does not exceed a defined limitation. This limitation may be either due to the total number of items output, the time during which items have been output, or the total size of all the items output. These limits are defined by way of the *Maximum Count*, *Maximum Time* and *Maximum Length* data stores. If a limitation is reached, then *Stop Timers* is indicated.

```
Inputs:
    In_Data_Length: Integer
    Maximum_Time: Real
    Maximum_Length: Integer
    Maximum_Count: Integer
Outputs:
    Data_Length: Integer
    Stop_Timers: SIGNAL
Processing:
    1. IF Maximum_Time < CURRENT_TIME THEN
        1. Maximum_Count := Maximum_Count - 1
        2. IF Maximum_Count > 0 THEN
            1. Maximum_Length := Maximum_Length - In_Data_Length
            2. IF Maximum_Length < 0
                1. In_Data_Length := In_Data_Length + Maximum_Length
            3. IF Maximum_Length > 0
                1. Data_Length := In_Data_Length
                2. STOP
    1. SIGNAL Stop_Timers
    2. STOP
```

PSPEC 3.3.1. Generate Telnet Profile

In *Generate Telnet Profile*, a *Data Length* and time interval, via. *Timer Activate*, are constructed using the TCPLIB samples that are provided.

```
Inputs:
    Next: SIGNAL
Outputs:
    Timer_Activate: SIGNAL
    Data_Length: Integer
Processing:
    1. Data_Length := TCPLIB_Telnet_Get_Length ()
    2. Timer_Activate := SET_TIMER (TCPLIB_Telnet_Get_Duration ())
    3. STOP
```

PSPEC 3.4.1. Generate FTP Profile

In *Generate FTP Profile*, a *Data Length* and time interval, via. *Timer Activate*, are constructed using the TCPLIB samples that are provided.

```
Inputs:
    Next: SIGNAL
Outputs:
    Timer_Activate: SIGNAL
    Data_Length: Integer
Processing:
    1. Data_Length := TCPLIB_FTP_Get_Length ()
    2. Timer_Activate := SET_TIMER (TCPLIB_FTP_Get_Duration ())
    3. STOP
```

PSPEC 3.5.1. Process Statistical IE

When a *Setup Statistical IE* is received by *Process Statistical IE*, its content is extracted and placed into the *Time Parameter* and *Space Parameter* outputs.

```
Inputs:
    Setup_Stat_IE: Generator Setup Statistical IE
Outputs:
    Time_Parameter: Statistical Info
    Space_Parameter: Statistical Info
Processing:
    1. Time_Parameter := ExtractIE_G_Setup_Statistical
                          (Setup_Stat_IE, TIME)
    2. Space_Parameter := ExtractIE_G_Setup_Statistical
                          (Setup_Stat_IE, SPACE)
    3. STOP
```

PSPEC 3.5.2. Generate Statistical Profile

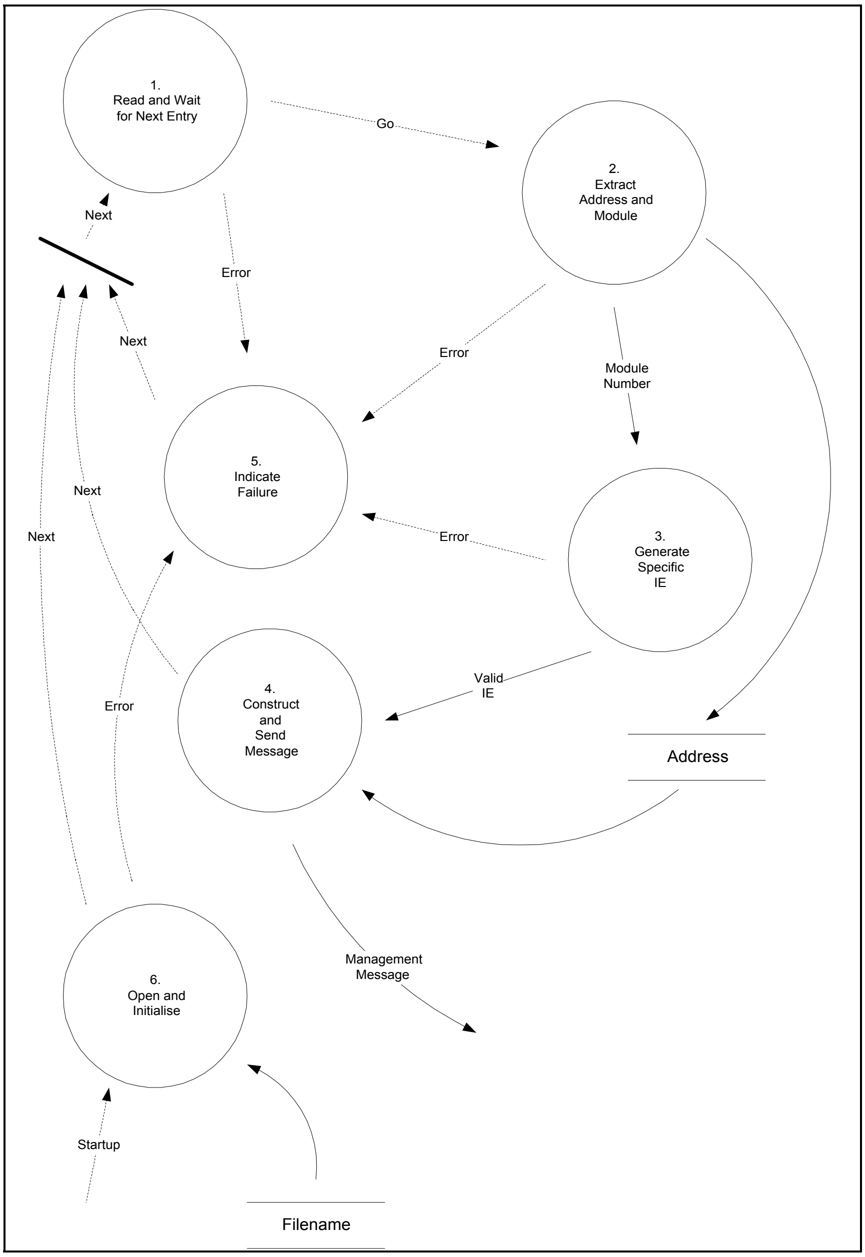
In *Generate Statistical Profile*, a *Data Length* and time interval, via. *Timer Activate*, are constructed using the Statistical module with the previously set *Time Parameter* and *Space Parameter*.

```
Inputs:
    Next: SIGNAL
    Time_Parameter: Statistical Info
    Space_Parameter: Statistical Info
Outputs:
    Timer_Activate: SIGNAL
    Data_Length: Integer
Processing:
    1. Data_Length := Get_Statistical_Info (Space_Parameter)
    2. Timer_Activate := SET_TIMER (Get_Statistical_Info
                          (Time_Parameter))
    3. STOP
```

2.8. Management

DFD 0: Top

The Top Level DFD illustrates the steps used to execute commands from *Filename*. *Open and Initialise* occurs on *Startup* after which there is a cycle of: *Read and Wait For Next Entry* (delay until the next entry is to be processed), *Extract Address and Module* (extract command's destination information), *Generate Specific IE* (process the command itself) *and Construct and Send Message* (transmit the IE to the specified destination). Any *Error* will *Indicate Failure*.



PSPEC 1. Read and Wait for Next Entry

In *Read and Wait for Next Entry*, the time at which the command is to occur is read; execution is suspended until that time is reached.

```
Inputs:
    Next: SIGNAL
Outputs:
    Error: SIGNAL
    Go: SIGNAL
Operation:
    1. DECLARE Time: Real
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. Time := File_Get_Real ()
    4. DELAY (Time - CURRENT_TIME)
    5. SIGNAL Go
    6. STOP
```

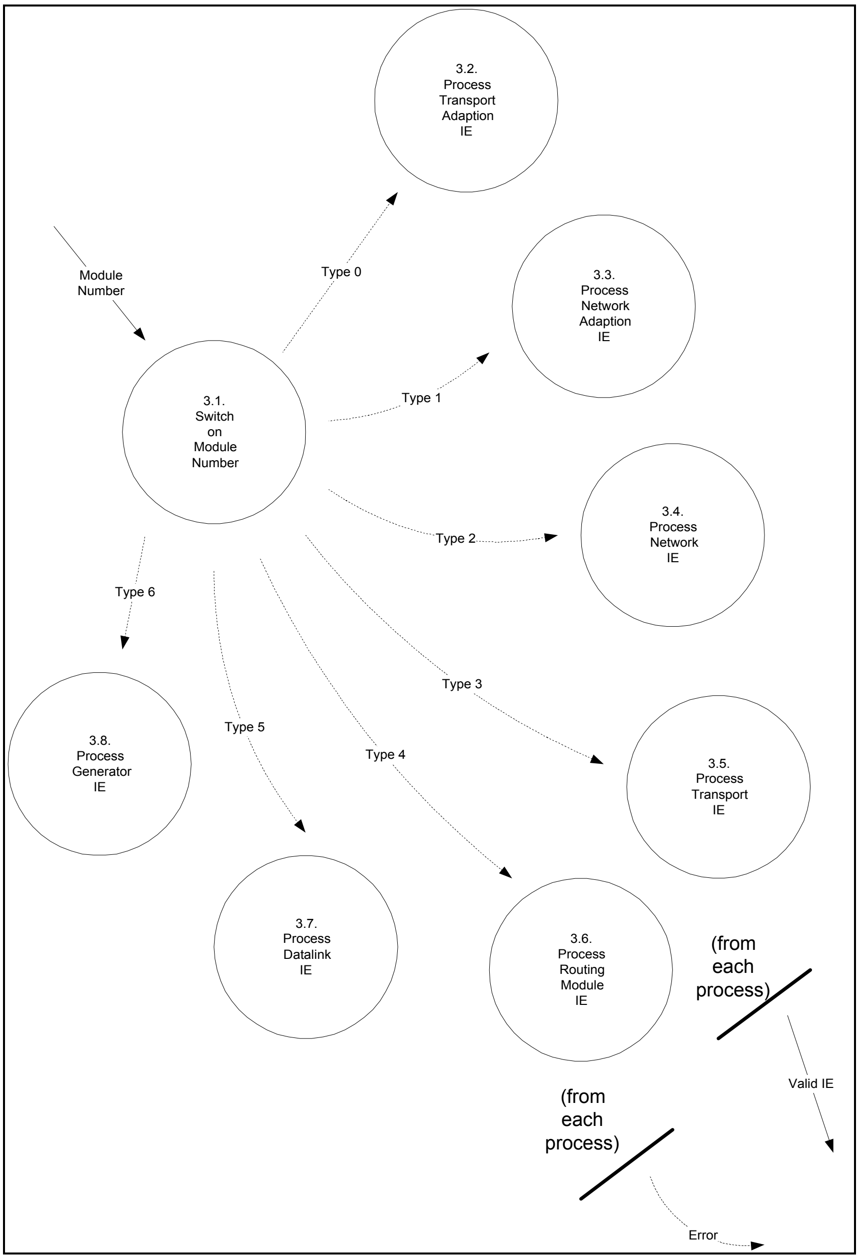
PSPEC 2. Extract Address and Module

In *Extract Address and Module*, the specific destination *Address* is read and stored, and the *Module Number* is also retrieved and passed on.

```
Inputs:
    Go: SIGNAL
Outputs:
    Module_Number: Integer
    Address: Integer
    Error: SIGNAL
Operation:
    1. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    2. Address := File_Get_Integer ()
    3. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    4. Module_Number := File_Get_Integer ()
    5. STOP
```

DFD 3. Generate Specific IE

In *Generate Specific IE*, the type of IE to create is determined by first passing processing onto a process depending on the given *Module Number*. Each of these processes is specific to a module type, and processing either results in a *Valid IE* or an *Error*.



PSPEC 4. Construct and Send Message

In *Construct and Send Message,* the given *Address* and a *Valid IE* are used to build a *Management Message*.

```
Inputs:
    Address: Integer
    Valid_IE: IE
Outputs:
    Management_Message: Management Set Indication Message
    Next: SIGNAL
Operation:
    1. Management_Message :=
            ConstructMsg_M_Set_Ind (Address, Valid_IE)
    2. SIGNAL Next
    3. STOP
```

PSPEC 5. Indicate Failure

In *Indicate Failure*, which occurs if some other failure occurred usually due to a premature end of file, or unexpected file contents, an error message is logged.

```
Inputs:
    Error: SIGNAL
Outputs:
    Next: SIGNAL
Operation:
    1. LOG ("Invalid File Contents")
    2. SIGNAL Next
    3. STOP
```

PSPEC 6. Open and Initialise

Open and Initialise concerns itself with using the *Filename* to open a file stream for subsequent use.

```
Inputs:
    Startup: SIGNAL
    Filename: String
Outputs:
    Error: SIGNAL
    Next: SIGNAL
Operation:
    1. IF File_Open (Filename) = Error THEN
        1. SIGNAL Error
        2. STOP
    2. SIGNAL Next
    3. STOP
```

PSPEC 3.1. Switch on Module Number

In *Switch on Module Number* the input *Module Number* is used to activate a specific process to parse the specified IE. Note that if the specified module number is incorrect, an *Error* will occur.

```
Inputs:
    Module_Number: Integer
Outputs:
    Type_0: SIGNAL
    Type_1: SIGNAL
    Type_2: SIGNAL
    Type_3: SIGNAL
    Type_4: SIGNAL
    Type_5: SIGNAL
    Type_6: SIGNAL
    Error: SIGNAL
Operation:
    1. IF Module_Number = 0 THEN
        1. SIGNAL Type_0
    2. IF Module_Number = 1 THEN
        1. SIGNAL Type_1
    3. IF Module_Number = 2 THEN
        1. SIGNAL Type_2
    4. IF Module_Number = 3 THEN
        1. SIGNAL Type_3
    5. IF Module_Number = 4 THEN
        1. SIGNAL Type_4
    6. IF Module_Number = 5 THEN
        1. SIGNAL Type_5
    7. IF Module_Number = 6 THEN
        1. SIGNAL Type_6
    8. IF Module_Number < 0 OR Module_Number > 6 THEN
        1. SIGNAL Error
    9. STOP
```

PSPEC 3.2. Process Transport-Adaption IE

In *Process Transport-Adaption IE* there are two possible commands that need to be interpreted into IEs. The first is a *Connect IE* and the second is a *Disconnect IE*. An *Error* will occur if the type to be interpreted is not known.

```
Inputs:
    Input_Kick: SIGNAL
Outputs:
    Error: SIGNAL
    Valid_IE: IE
Operation:
    1. DECLARE Type: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. Type := File_Get_Integer ()
    4. IF Type = 0 THEN
        1. (Error, IE) := Process_Connect_IE ()
        2. STOP
    5. IF Type = 1 THEN
        1. (Error, IE) := Process_Disconnect_IE ()
        2. STOP
    6. SIGNAL Error
    7. STOP
```

FUNCTION 3.2.1. Process Connect IE

The *Process_Connect_IE* consists of reading in a single *Address* to be placed into the IE.

```
Function:
    Process_Connect_IE
Inputs:
    n/a
Outputs:
    Error: SIGNAL
    Connect_IE: Transport-Adaption Connect IE
Operation:
    1. DECLARE Address: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. Address := File_Get_Integer ()
    4. Connect_IE := ConstructIE_TA_Connect (Address)
    5. STOP
```

FUNCTION 3.2.2. Process Disconnect IE

The *Process_Disconnect_IE* consists only of the construct of an IE as there are no parameters to be placed into the IE.

```
Function:
    Process_Disconnect_IE
Inputs:
    n/a
Outputs:
    Error: SIGNAL
    Disconnect_IE: Transport-Adaption Disconnect IE
Operation:
    1. Disconnect_IE := ConstructIE_TA_Disconnect ()
    2. STOP
```

PSPEC 3.3. Process Network-Adaption IE

In *Process Network-Adaption IE* there is only one possible command that needs to be interpreted into an IE. This is the *Address List IE*. An *Error* will occur if the type is not known.

```
Inputs:
    Input_Kick: SIGNAL
Outputs:
    Error: SIGNAL
    Valid_IE: IE
Operation:
    1. DECLARE Type: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. Type := File_Get_Integer ()
    4. IF Type = 0 THEN
        1. (Error, IE) := Process_Address_List_IE ()
        2. STOP
    5. SIGNAL Error
    6. STOP
```

FUNCTION 3.3.1. Process Address List IE

In *Process Address List IE*, the first entry is read to indicate how many Addresses will be present, then each one is subsequently read in and placed into the *Address List*. The IE is created using this *Address List*.

```
Function:
    Process_Address_List_IE
Inputs:
    n/a
Outputs:
    Error: SIGNAL
    Address_List_IE: Network-Adaption Address List IE
Operation:
    1. DECLARE Address_List: ARRAY OF Integer
    2. DECLARE Count: Integer
    3. DECLARE Address_Count: Integer
    4. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    5. Address_Count := File_Get_Integer ()
    6. Count := 0
    Label_Loop_Next:
    7. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    8. Address_List[Count] := File_Get_Integer ()
    9. Count := Count + 1
    10. IF Count < Address_Count THEN GOTO Label_Loop_Next
    11. Address_List_IE :=
            ConstructIE_NA_Address_List (Address_List)
    12. STOP
```

PSPEC 3.4. Process Network IE

There are no IEs currently defined for the Network Layer, so *Process Network IE* is a stub to be expanded at a later date. As such, it currently generates an error on any invocation.

```
Inputs:
    Input_Kick: SIGNAL
Outputs:
    Error: SIGNAL
    Valid_IE: IE
Operation:
    1. SIGNAL Error
    2. STOP
```

PSPEC 3.5. Process Transport IE

In *Process Transport IE* there is only one possible command that needs to be interpreted into an IE. This is the *Setup IE*. An *Error* will occur if the type is not known.

```
Inputs:
    Input_Kick: SIGNAL
Outputs:
    Error: SIGNAL
    Valid_IE: IE
Operation:
    1. DECLARE Type: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. Type := File_Get_Integer ()
    4. IF Type = 0 THEN
        1. (Error, IE) := Process_Setup_IE ()
        2. STOP
    5. SIGNAL Error
    6. STOP
```

FUNCTION 3.5.1. Process Setup IE

In *Process Setup IE*, there is one parameter to be read, and this is the *ISN*. This is used in the creation of the IE.

```
Function:
    Process_Setup_IE
Inputs:
    n/a
Outputs:
    Error: SIGNAL
    Setup_IE: Transport Setup IE
Operation:
    1. DECLARE ISN: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. ISN := File_Get_Integer ()
    4. Setup_IE := ConstructIE_T_Setup (ISN)
    5. STOP
```

PSPEC 3.6. Process Routing-Module IE

In *Process Routing-Module IE* there is only one possible command that needs to be interpreted into an IE. This is the *Routing Entry IE.* An *Error* will occur if the type is not known.

```
Inputs:
    Input_Kick: SIGNAL
Outputs:
    Error: SIGNAL
    Valid_IE: IE
Operation:
    1. DECLARE Type: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. Type := File_Get_Integer ()
    4. IF Type = 0 THEN
        1. (Error, IE) := Process_Routing_Entry_IE ()
        2. STOP
    5. SIGNAL Error
    6. STOP
```

FUNCTION 3.6.1. Process Routing Entry IE

In *Process Routing Entry IE*, there are three parameters to be read, the *Address*, *Interface* and *Cost*. These are used in the creation of the IE.

```
Function:
    Process_Routing_Entry_IE
Inputs:
    n/a
Outputs:
    Error: SIGNAL
    Routing_Entry_IE: Routing-Module Routing Entry IE
Operation:
    1. DECLARE Address: Integer
    2. DECLARE Interface: Integer
    3. DECLARE Cost: Real
    4. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    5. Address := File_Get_Integer ()
    6. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    7. Interface := File_Get_Integer ()
    8. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    9. Cost := File_Get_Real ()
    10. Routing_Entry_IE :=
           ConstructIE_R_Routing_Entry (Address, Interface, Cost)
    11. STOP
```

PSPEC 3.7. Process Datalink IE

In *Process Datalink IE* there is only one possible command that needs to be interpreted into an IE. This is the *State IE*. An *Error* will occur if the type is not known..

```
Inputs:
    Input_Kick: SIGNAL
Outputs:
    Error: SIGNAL
    Valid_IE: IE
Operation:
    1. DECLARE Type: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. Type := File_Get_Integer ()
    4. IF Type = 0 THEN
        1. (Error, IE) := Process_State_IE ()
        2. STOP
    5. SIGNAL Error
    6. STOP
```

FUNCTION 3.7.1. Process State IE

In *Process State IE*, there is one parameter to be read. This parameter is the *State* for the Datalink Layer and is used in the creation of the IE.

```
Function:
    Process_State_IE
Inputs:
    n/a
Outputs:
    Error: SIGNAL
    State_IE: Datalink State IE
Operation:
    1. DECLARE State: Boolean
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. State := File_Get_Boolean ()
    4. State_IE := ConstructIE_DL_State (State)
    5. STOP
```

PSPEC 3.8. Process Generator IE

In *Process Generator IE* there are two possible commands that need to be interpreted into IEs. The first is a *Setup IE* and the second is a *Stop IE*. An *Error* will occur if the type to be interpreted is not known.

```
Inputs:
    Input_Kick: SIGNAL
Outputs:
    Error: SIGNAL
    Valid_IE: IE
Operation:
    1. DECLARE Type: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. Type := File_Get_Integer ()
    4. IF Type = 0 THEN
        1. (Error, IE) := Process_Setup_IE ()
        2. STOP
    5. IF Type = 1 THEN
        1. (Error, IE) := Process_Stop_IE ()
        2. STOP
    6. SIGNAL Error
    7. STOP
```

FUNCTION 3.8.1. Process Stop IE

The functionality for *Process Stop IE* is simple in that there are no parameters, so only the construction of the IE occurs.

```
Function:
    Process_Stop_IE
Inputs:
    n/a
Outputs:
    Error: SIGNAL
    Stop_IE: Generator Stop IE
Operation:
    1. Stop_IE := ConstructIE_G_Stop ()
    2. STOP
```

FUNCTION 3.8.2. Process Setup IE

The functionality for *Process Setup IE* is slightly more complex in that there are three global filter parameters, followed by specific parameters according to the type of generation that will occur, within which there may be more parameters.

```
Function:
    Process_Setup_IE
Inputs:
    n/a
Outputs:
    Error: SIGNAL
    Setup_IE: Generator Setup IE
Operation:
    1. DECLARE Count: Integer
    2. DECLARE Time: Real
    3. DECLARE Length: Integer
    4. DECLARE Type: Integer
    5. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    6. Count := File_Get_Integer ()
    7. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    8. Time := File_Get_Real ()
    9. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    10. Length := File_Get_Integer ()
    11. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    12. Type := File_Get_Integer ()
    13. IF Type = 0 THEN
        1. Setup_IE :=
                   ConstructIE_G_Setup_Telnet (Count, Time, Length)
    14. IF Type = 1 THEN
        1. Setup_IE := ConstructIE_G_Setup_FTP (Count, Time, Length)
    15. IF Type = 2 THEN
        1. DECLARE Time_Stat: Statistical Info
        2. DECLARE Space_Stat: Statistical Info
        3. (Time_Stat, Error) := Process_Stat_Info ()
        4. IF Error THEN
            1. STOP
        5. (Space_Stat, Error) := Process_Stat_Info ()
        6. IF Error THEN
            1. STOP
        7. Setup_IE := ConstructIE_G_Setup_Statistical (Count,
                   Time, Length, Stat)
    16. STOP
```

FUNCTION 3.8.3. Process Stat Info

```
Function:
    Process_Stat_Info
Inputs:
    n/a
Outputs:
    Stat: Statistical Info
    Error: SIGNAL
Operation:
    1. DECLARE Type: Integer
    2. IF File_Is_End () = True THEN
        1. SIGNAL Error
        2. STOP
    3. IF Type = 0 THEN
        1. DECLARE Value: Real
        2. IF File_Is_End () = True THEN
            1. SIGNAL Error
            2. STOP
        3. Value := File_Get_Real ()
        4. Stat := Create_Stat_Constant (Value)
    4. IF Type = 1 THEN
        1. DECLARE Lower: Real
        2. DECLARE Upper: Real
        3. IF File_Is_End () = True THEN
            1. SIGNAL Error
            2. STOP
        4. Lower := File_Get_Real ()
        5. IF File_Is_End () = True THEN
            1. SIGNAL Error
            2. STOP
        6. Upper := File_Get_Real ()
        7. Stat := Create_Stat_Uniform (Lower, Upper)
    5. IF Type = 2 THEN
        1. DECLARE Mean: Real
        2. DECLARE Variance: Real
        3. IF File_Is_End () = True THEN
            1. SIGNAL Error
            2. STOP
        4. Mean := File_Get_Real ()
        5. IF File_Is_End () = True THEN
            1. SIGNAL Error
            2. STOP
        6. Variance := File_Get_Real ()
        7. Stat := Create_Stat_Normal (Mean, Variance)
    6. IF Type = 3 THEN
        1. DECLARE Mean: Real
        2. IF File_Is_End () = True THEN
            1. SIGNAL Error
            2. STOP
        3. Mean := File_Get_Real ()
        4. Stat := Create_Stat_Exponential (Mean)
    7. IF Type = 4 THEN
        1. DECLARE Lambda: Real
        2. IF File_Is_End () = True THEN
            1. SIGNAL Error
            2. STOP
        3. Lambda := File_Get_Real ()
        4. Stat := Create_Stat_Poisson (Lambda)
    8. IF Type < 0 OR Type > 4 THEN
        1. SIGNAL Error
    9. STOP
```

3. Miscellaneous Modules

3.1. Statistical Parameter

There are two classes of functions. The first class has one member, this is Get_Statistical_Info, and its purpose is to create an instance value of the parameter, without knowing what particular parameter it is that is being created. Internally, the creation is deferred to a subprocedure applicable to the type of parameter that there is:

```
Function:
    Get_Statistical_Info
Inputs:
    Info: Statistical Info
Outputs:
    Value: REAL
Operation:
    1. IF Type (Info) = 'Statistical Info Constant' THEN
        1. Value := Get_Statistical_Info_Constant (Info)
    2. IF Type (Info) = 'Statistical Info Uniform' THEN
        1. Value := Get_Statistical_Info_Uniform (Info)
    3. IF Type (Info) = 'Statistical Info Normal' THEN
        1. Value := Get_Statistical_Info_Normal (Info)
    4. IF Type (Info) = 'Statistical Info Exponential' THEN
        1. Value := Get_Statistical_Info_Exponential (Info)
    5. IF Type (Info) = 'Statistical Info Poisson' THEN
        1. Value := Get_Statistical_Info_Poisson (Info)
    6. STOP
```

```
Function:
    Get_Statistical_Info_Constant
Inputs:
    Info: Statistical Info Constant
Outputs:
    Value: REAL
Operation:
    1. Value := Info.Value
    2. STOP
```

```
Function:
    Get_Statistical_Info_Uniform
Inputs:
    Info: Statistical Info Uniform
Outputs:
    Value: REAL
Operation:
    1. Value := RANDOM_UNIFORM (Info.Minimum, Info.Maximum)
    2. STOP
```

```
Function:
    Get_Statistical_Info_Normal
Inputs:
    Info: Statistical Info Normal
Outputs:
    Value: REAL
Operation:
    1. Value := RANDOM_NORMAL (Info.Mean, Info.Variance)
    2. STOP
```

```
Function:
    Get_Statistical_Info_Exponential
Inputs:
    Info: Statistical Info Exponential
Outputs:
    Value: REAL
Operation:
    1. Value := RANDOM_EXP (Info.Mean)
    2. STOP
```

```
Function:
    Get_Statistical_Info_Poisson
Inputs:
    Info: Statistical Info Poisson
Outputs:
    Value: REAL
Operation:
    1. Value := RANDOM_POISSON (Info.Lambda)
    2. STOP
```

The second class of functions are creators; they allow for the creation of a specific type of statistical parameter and return the abstract data structure upon return. There is an implicit mechanism in here that determines the types that is not shown.

```
Function:
    Create_Stat_Constant
Inputs:
    Value: REAL
Outputs:
    Info: Statistical Info
Operation:
    1. Info.Value := Value
    2. STOP
```

```
Function:
    Create_Stat_Uniform
Inputs:
    Minimum: REAL
    Maximum: REAL
Outputs:
    Info: Statistical Info
Operation:
    1. Info.Minimum := Minimum
    2. Info.Maximum := Maximum
    2. STOP
```

```
Function:
    Create_Stat_Normal
Inputs:
    Mean: REAL
    Variance: REAL
Outputs:
    Info: Statistical Info
Operation:
    1. Info.Mean := Mean
    2. Info.Variance := Variance
    3. STOP
```

```
Function:
    Create_Stat_Exponential
Inputs:
    Mean: REAL
Outputs:
    Info: Statistical Info
Operation:
    1. Info.Mean := Mean
    2. STOP
```

```
Function:
    Create_Stat_Poisson
Inputs:
    Lambda: REAL
Outputs:
    Info: Statistical Info
Operation:
    1. Info.Lambda := Lambda
    2. STOP
```

3.2. Transport Layer -- TCP Probe

The TCP Probe maintains a Table of Parameters and Functions that it uses to carry out its operation. It determines which function is to be used, based upon the supplied Data Type Parameter, and then for each execution, it attempts to obtain the data value from that function.

```
DEF TABLE Probe_Functions (Integer: Index,
              String: Parameter,
              Function: Processor)
    0,  "Congestion Window",           Get_Congestion_Window
    1,  "Slow Start Threshold",        Get_Slow_Start_Threshold
    2,  "Retransmission Events",       Get_ReTx_Events
    3,  "Round Trip Time Average",     Get_RTT_Average
    4,  "Round Trip Time Variance",    Get_RTT_Variance
    5,  "Send Window",                 Get_Send_Window
    6,  "Unacknowledged Data",         Get_Unacknowledged_Data
    7,  "Timer Expries",               Get_Timer_Expiries
    8,  "Acknowledgements Received",   Get_Ack_Received
    9,  "KB Retransmitted",            Get_KB_ReTx
    10, "KB Transmitted",              Get_KB_Tx
    11, "Reassembly Queue Size",       Get_Reassembly_Queue_Size
ENDDEF
```

```
Function:
    TCP_Probe_Init
Inputs:
    First_Value: Boolean
    Data_Type: String
Outputs:
    Table_Index: Integer
Processing
    1. First_Value := True
    2. Table_Index := SELECT   Index
                      FROM     Probe_Functions
                      WHERE    Parameter = Data_Type
    3. STOP
```

```
Function:
    TCP_Probe_Execute
Inputs:
    TCB_Index: Integer
    Duplicate: Boolean
    First_Value: Boolean
    Old_Value: Real
Outputs:
    New_Value: Real
Processing:
    1. DECLARE Tcb: TcbPtr
    2. DECLARE New_Value: Real
    3. Tcb := TCB_Lookup (TCB_Index);
    4. New_Value = TABLE (Table_Index).Processor (Tcb);
    5. IF (Duplicate == FALSE OR New_Value != Old_Value OR
            First_Value == TRUE) THEN
        1. OUTPUT (New_Value)
        2. First_Value = FALSE;
        3. Old_Value = New_Value
    6. ENDIF
    7. STOP
```

Each particular function is implemented as:

```
Function:
    Get_Congestion_Window
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.snd_cwnd
    2. Success := True
    3. STOP
```

```
Function:
    Get_Slow_Start_Threshold
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.snd_ssthresh
    2. Success := True
    3. STOP
```

```
Function:
    Get_ReTx_Events
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.probe_retx_count
    2. Tcb.probe_retx_count = 0
    3. IF Value > 0 THEN
        1. Success := True
    4. ELSE
        1. Success := False
    5. STOP
```

```
Function:
    Get_RTT_Average
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.t_srtt
    2. Success := True
    3. STOP
```

```
Function:
    Get_RTT_Variance
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.rttvar
    2. Success := True
    3. STOP
```

```
Function:
    Get_Send_Window
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.snd_wnd
    2. Success := True
    3. STOP
```

```
Function:
    Get_Unacknowledged_Data
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.snd_wnd - (Tcb.snd_nxt - Tcb.snd_una)
    2. Success := True
    3. STOP
```

```
Function:
    Get_Timer_Expiries
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.probe_texpiry
    2. Tcb.probe_texpiry = 0
    3. IF Value > 0 THEN
        1. Success := True
    4. ELSE
        1. Success := False
    5. STOP
```

```
Function:
    Get_Ack_Received
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.probe_ackrecv
    2. Tcb.probe_ackrecv = 0
    3. IF Value != 0 THEN
        1. Success := True
    4. ELSE
        1. Success := False
    5. STOP
```

```
Function:
    Get_KB_ReTx
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.probe_retx_count
    2. Success := True
    3. STOP
```

```
Function:
    Get_KB_Tx
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Tcb.probe_tx_count
    2. Success := True
    3. STOP
```

```
Function:
    Get_Reassembly_Queue_Size
Inputs:
    Tcb: TcbPtr
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := QueueSize (Tcb.FragmentQueue)
    2. Success := True
    3. STOP
```

3.3. Network Layer -- Queue Probe

The Queue Probe maintains a Table of Parameters and Functions that it uses to carry out its operation. It determines which function is to be used, based upon the supplied Data Type Parameter, and then for each execution, it attempts to obtain the data value from that function.

```
DEF TABLE Probe_Functions (Integer: Index,
              String: Parameter,
              Function: Processor)
    0,  "Size",                          Get_Size
    1,  "Source Address Count",          Get_Src_Address_Count
    2,  "Dest Address Count",            Get_Src_Address_Count
ENDDEF
```

```
Function:
    Queue_Probe_Init
Inputs:
    First_Value: Boolean
    Data_Type: String
Outputs:
    Table_Index: Integer
Processing
    1. First_Value := True
    2. Table_Index := SELECT   Index
                      FROM     Probe_Functions
                      WHERE    Parameter = Data_Type
    3. STOP
```

```
Function:
    Queue_Probe_Execute
Inputs:
    Queue_Index: Integer
    Duplicate: Boolean
    First_Value: Boolean
    Old_Value: Real
    Address: Integer
Outputs:
    New_Value: Real
Processing:
    1. DECLARE Queue: Queue Entry
    2. DECLARE New_Value: Real
    3. Queue := Queue_Lookup (Queue_Index);
    3. GLOBAL Address = Address
    4. New_Value = TABLE (Table_Index).Processor (Queue);
    5. IF (Duplicate == FALSE OR New_Value != Old_Value OR
            First_Value == TRUE) THEN
        1. OUTPUT (New_Value)
        2. First_Value = FALSE;
        3. Old_Value = New_Value
    6. ENDIF
    7. STOP
```

And the individual functions are.

```
Function:
    Get_Size
Inputs:
    Queue: QueueEntry
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := Get_Size (Queue)
    2. Success := True
    3. STOP
```

```
Function:
    Get_Src_Address_Count
Inputs:
    Queue: QueueEntry
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := 0
    2. FOREACH Item IN Queue DO
        1. IF Get_Source_Address (Element) = GLOBAL Address THEN
            1. Value := Value + 1
        2. ENDIF
    3. ENDDO
    4. Success := True
    5. STOP
```

```
Function:
    Get_Dst_Address_Count
Inputs:
    Queue: QueueEntry
Outputs:
    Value: REAL
    Success: Boolean
Operation:
    1. Value := 0
    2. FOREACH Item IN Queue DO
        1. IF Get_Dest_Address (Element) = GLOBAL Address THEN
            1. Value := Value + 1
        2. ENDIF
    3. ENDDO
    4. Success := True
    5. STOP
```

APPENDIX 2. DETAILED BONES IMPLEMENTATION

1. Overview

The following sections provide the detailed aspects of the BONeS implementation. This consists of a breakdown of all Modules in terms of their Data Structures, Main Modules, Support Modules

The Data Structures are presented in tables, providing a verbose indication of constituent fields including those that are inherited from parent Data Structures (represented in italics). For Modules, BONeS diagrams are used to illustrate their construction, and ‘C’ source code is provided where any such implementation was carried out. Each Module has much more information, in terms of ports, parameters and so on -- inclusion of this information would tend to expand the already comprehensive information. It is important to document this information as it provides necessary details behind the design.

There is a lot of detail here, as the implementation was partitioned significantly (the intention to construct many small modules, rather than big unwieldy modules). In addition, the ‘C’ source code is also verbose, due to its nature of being so.

2. Primary Modules

2.1. Datalink Layer

2.1.1. Data Structures

2.1.1.1. IE Datalink Primitive

This Data Structure has no content.

2.1.1.2. IE Datalink Flow Control

Name	*Type*	*Subrange*	*Default Value*
Flow Control Released	Boolean		True

2.1.1.3. IE Datalink State

Name	*Type*	*Subrange*	*Default Value*
State	Boolean		True

2.1.1.4. Msg Datalink Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.1.1.5. Msg Datalink Connect Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.1.1.6. Msg Datalink Connect Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.1.1.7. Msg Datalink Data Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	Msg Primitive		

2.1.1.8. Msg Datalink Data Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	Msg Primitive		

2.1.1.9. Msg Datalink Data Request

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	Msg Primitive		

2.1.1.10. Msg Datalink Disconnect Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.1.1.11. Msg Datalink Disconnect Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.1.1.12. Msg Datalink Status Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	IE Datalink Primitive		

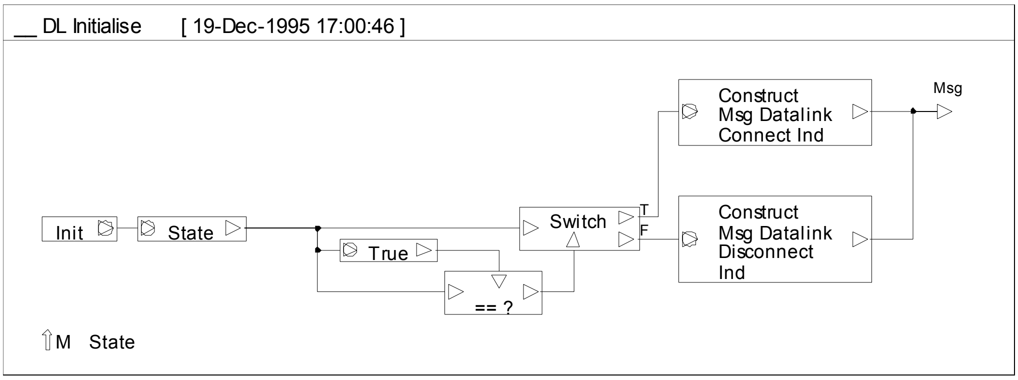
2.1.1.13. Msg Datalink Status Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	*IE Datalink Primitive*		

2.1.2. Main Modules

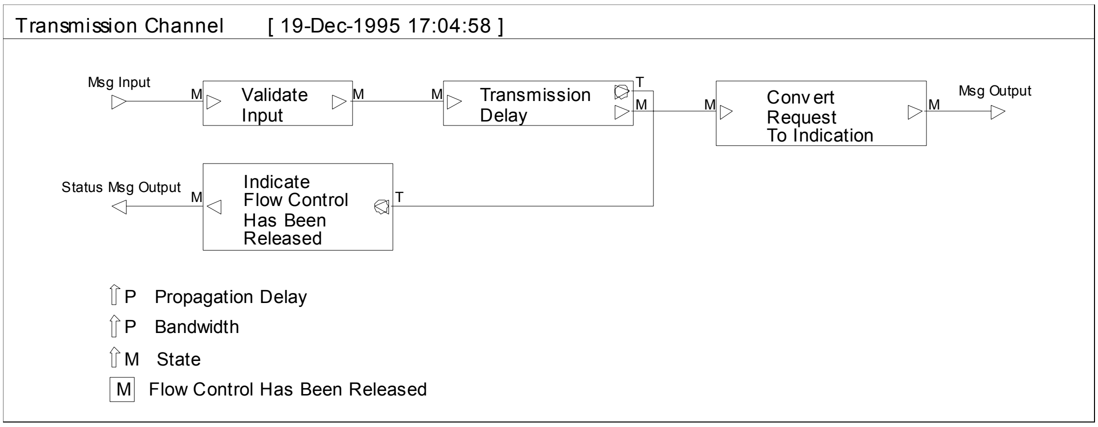
2.1.2.1. Initialisation

This module is not shown in the design, as the design assumed implicit initialisation whereas the initialisation is in fact explicit. It is responsible for setting the startup *State* of the Datalink Layer and generating an appropriate *Connect* or *Disconnect Indication* Message to its higher layer in order to inform it.


__ DL Initialise [19-Dec-1995 17:00:46]
Init
State
True
== ?
Switch
T
F
Construct Msg Datalink Connect Ind
Construct Msg Datalink Disconnect Ind
Msg
M State

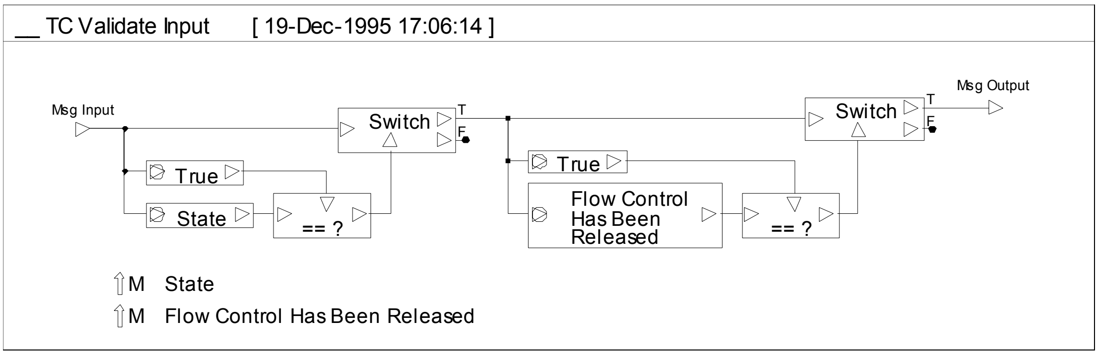
2.1.2.2. Transmission Channel

This Module is constructed for “DFD 1: Transmission Channel”, noting that that *Flow Control Has Been Released* is used in place of *Flow Control State*.


Transmission Channel [19-Dec-1995 17:04:58]
Msg Input
M
Validate Input
M
M
Transmission Delay
T
M
M
Convert Request To Indication
M
Msg Output
Status Msg Output
M
Indicate Flow Control Has Been Released
T
P Propagation Delay
P Bandwidth
M State
M Flow Control Has Been Released

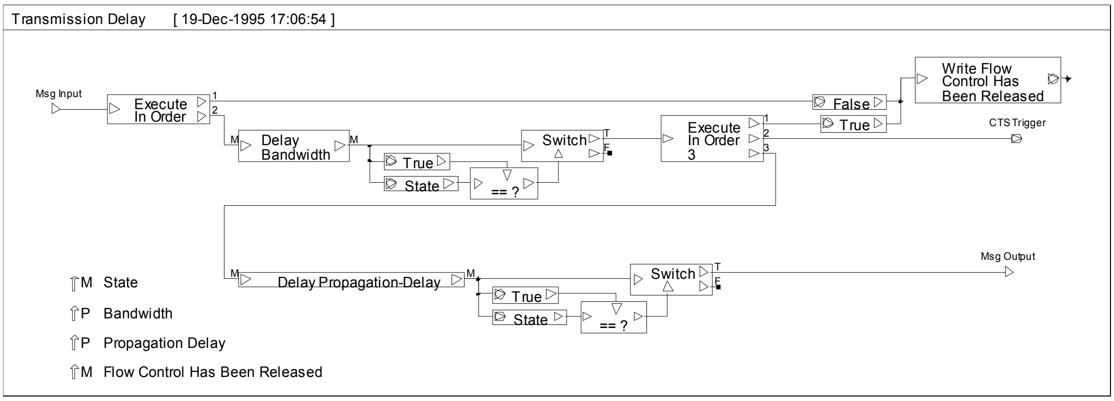
2.1.2.3. Transmission Channel -- Validate Input

This Module implements “PSPEC 1.1: Validate Input”.


__ TC Validate Input [19-Dec-1995 17:06:14]
Msg Input
Switch
T
F
True
State
== ?
True
Flow Control Has Been Released
== ?
Switch
T
F
Msg Output
M State
M Flow Control Has Been Released

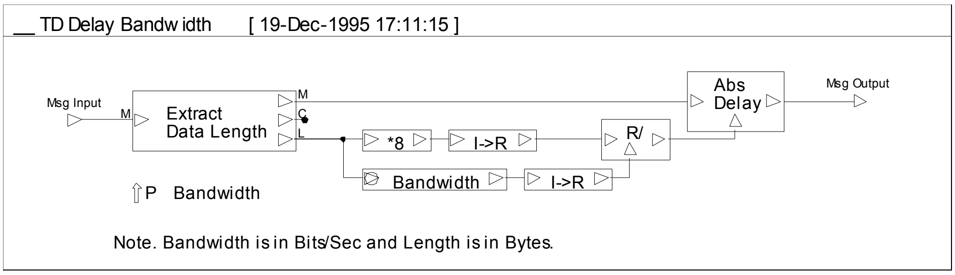
2.1.2.4. Transmission Channel -- Transmission Delay

This Module is constructed for “PSPEC 1.2: Execute Transmission Delay”. The two SLEEP functional points have been delegated to submodules (*Delay Bandwidth* and *Delay Propagation Delay*) to prevent unecessary cluttering at this level.



2.1.2.5. Transmission Channel -- Transmission Delay -- Delay Bandwidth

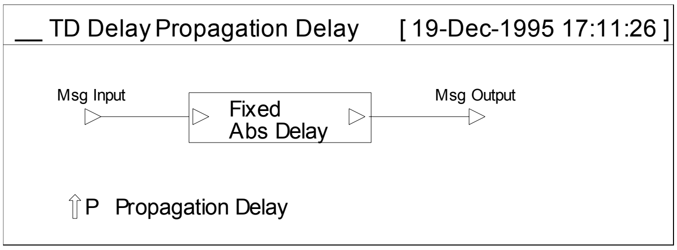
This Module implements part of “PSPEC 1.2: Execute Transmission Delay” and uses BONeS *Abs Delay* primitive to SLEEP for a time corresponding to the length of the Message.



Note. Bandwidth is in Bits/Sec and Length is in Bytes.

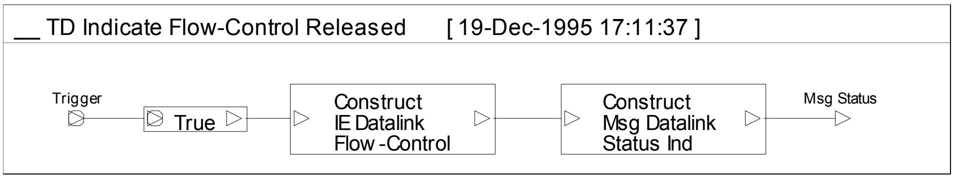
2.1.2.6. Transmission Channel -- Transmission Delay -- Delay Propagation Delay

This Module implements part of “PSPEC 1.2: Execute Transmission Delay” and uses BONeS *Fixed Abs Delay* to SLEEP for a time corresponding to the *Propagation Delay* parameter.



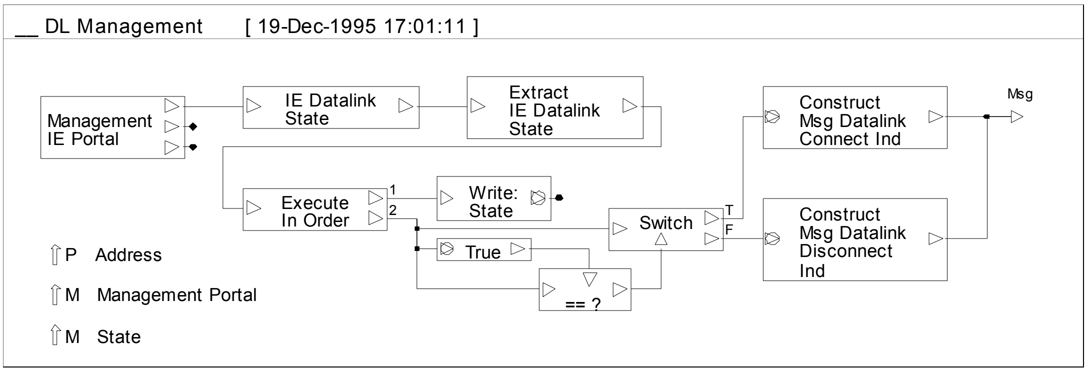
2.1.2.7. Transmission Channel -- Indicate Flow Control Released

This Module implements “PSPEC 1.3: Indicate Flow Control Status”.



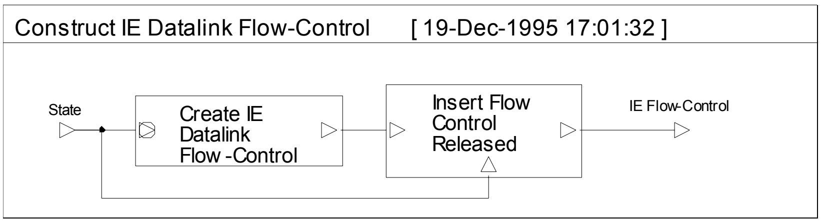
2.1.2.8. Management

This Module implements “DFD 2: Management Processor”, “PSPEC 2.1: Validate Mgmt Message and Extract IE” and “PSPEC 2.2: Process Status IE”. Note that the *Management Message* is received through the *Management IE Portal.* PSPEC 2.1 is subsumed by the *Management IE Portal* and PSPEC 2.2 has been aggregated for convenience.

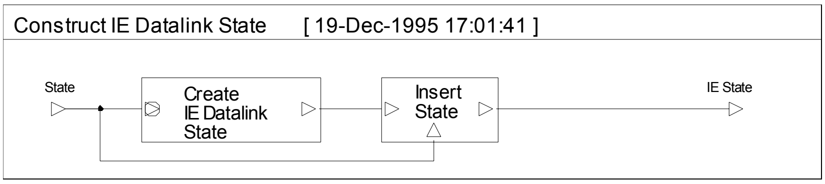


2.1.3. Support Modules

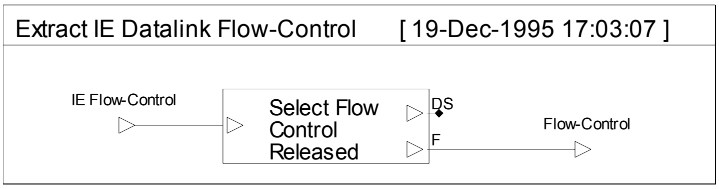
2.1.3.1. Construct IE Datalink Flow Control


Construct IE Datalink Flow-Control [19-Dec-1995 17:01:32]
State
Create IE Datalink Flow -Control
Insert Flow Control Released
IE Flow-Control

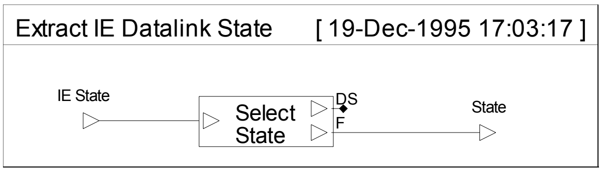
2.1.3.2. Construct IE Datalink State



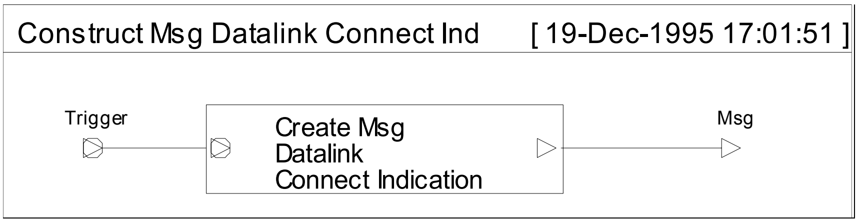
2.1.3.3. Extract IE Datalink Flow Control


Extract IE Datalink Flow-Control [19-Dec-1995 17:03:07]
IE Flow-Control
Select Flow Control Released
DS
F
Flow-Control

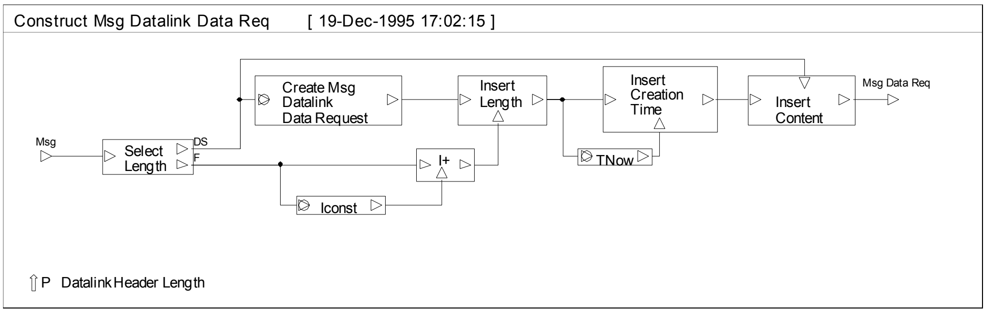
2.1.3.4. Extract IE Datalink State



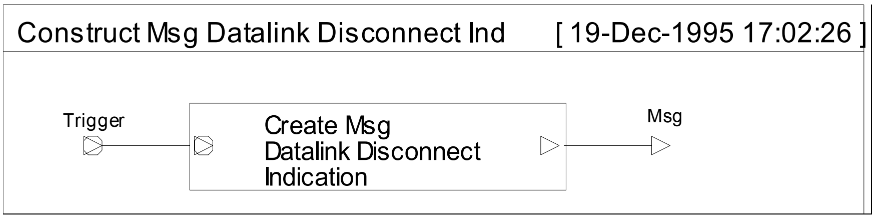
2.1.3.5. Construct Msg Datalink Connect Indication


Construct Msg Datalink Connect Ind [19-Dec-1995 17:01:51]
Trigger
Create Msg Datalink Connect Indication
Msg

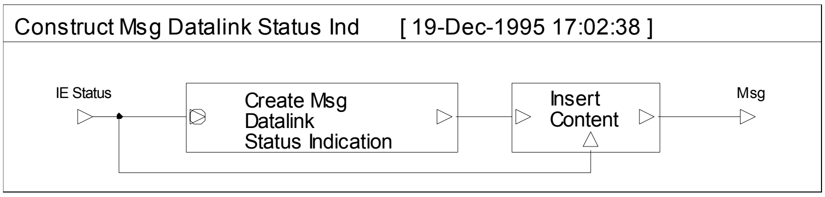
2.1.3.6. Construct Msg Datalink Data Request



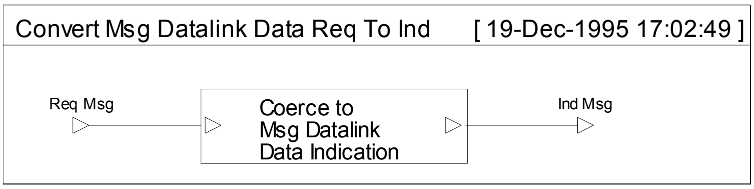
2.1.3.7. Construct Msg Datalink Disconnect Indication


Construct Msg Datalink Disconnect Ind [19-Dec-1995 17:02:26]
Trigger
Create Msg Datalink Disconnect Indication
Msg

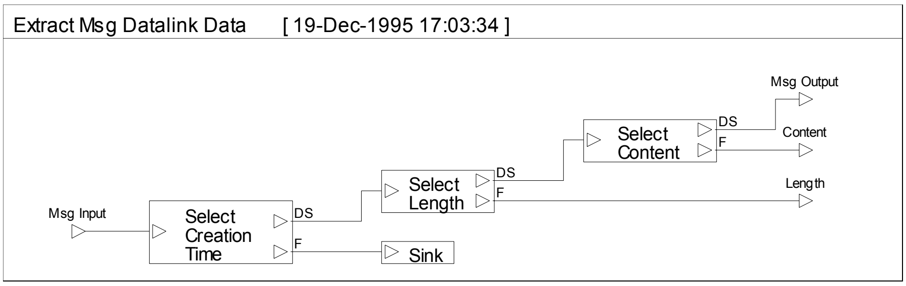
2.1.3.8. Construct Msg Datalink Status Indication


Construct Msg Datalink Status Ind [19-Dec-1995 17:02:38]
IE Status
Create Msg Datalink Status Indication
Insert Content
Msg

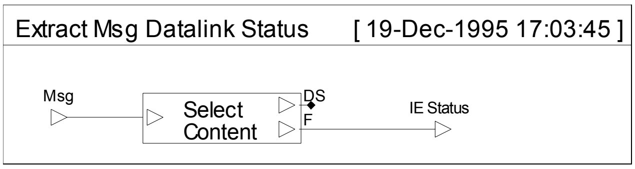
2.1.3.9. Convert Msg Datalink Data Request to Indication


Convert Msg Datalink Data Req To Ind [19-Dec-1995 17:02:49]
Req Msg
Coerce to Msg Datalink Data Indication
Ind Msg

2.1.3.10. Extract Msg Datalink Data



2.1.3.11. Extract Msg Datalink Status



2.2. Network Layer

2.2.1. Data Structures

2.2.1.1. IE Network Primitive

This Data Structure has no content.

2.2.1.2. IE Network Load-Factor

Name	*Type*	*Subrange*	*Default Value*
Load Factor	REAL	[0.0,1.0]	0.0

2.2.1.3. Msg Network Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.2.1.4. Msg Network Connect Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.2.1.5. Msg Network Connect Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.2.1.6. Msg Network Data Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Destination Address	INTEGER	[0,512)	0
Hop Count	INTEGER	[0,256)	255
Explicit Congestion Notification	INTEGER	[0,+Inf)	0
Source Address	INTEGER	[0,512)	0
Content	Msg Primitive		

2.2.1.7. Msg Network Data Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Destination Address	*INTEGER*	*[0,512)*	*0*
Hop Count	*INTEGER*	*[0,256)*	*255*
Explicit Congestion Notification	*INTEGER*	*[0,+Inf)*	*0*
Source Address	*INTEGER*	*[0,512)*	*0*
Content	*Msg Primitive*		

2.2.1.8. Msg Network Data Request

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Destination Address	*INTEGER*	*[0,512)*	*0*
Hop Count	*INTEGER*	*[0,256)*	*255*
Explicit Congestion Notification	*INTEGER*	*[0,+Inf)*	*0*
Source Address	*INTEGER*	*[0,512)*	*0*
Content	*Msg Primitive*		

2.2.1.9. Msg Network Disconnect Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.2.1.10. Msg Network Disconnect Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.2.1.11. Msg Network Status Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	IE Network Primitive		

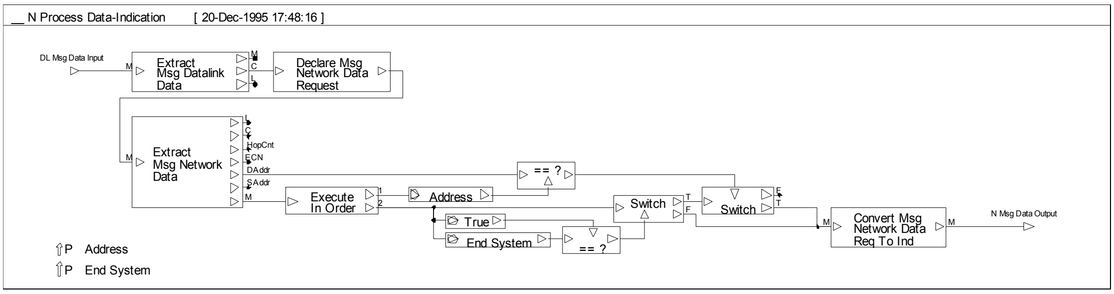
2.2.1.12. Msg Network Status Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	*IE Network Primitive*		

2.2.2. Main Modules

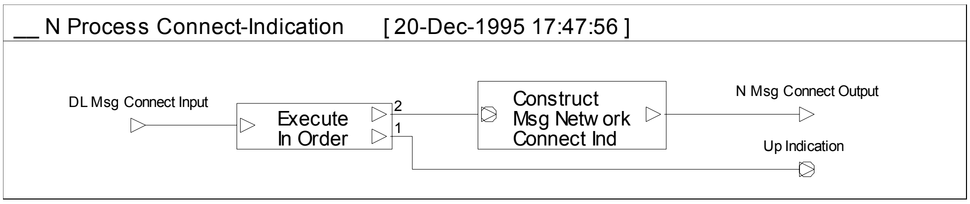
2.2.2.1. Process Data Indication

This Module implements “PSPEC 1.2: Process Data Message”. Note that “DFD 1: Process Datalink Message” has been subsumed by the Top.



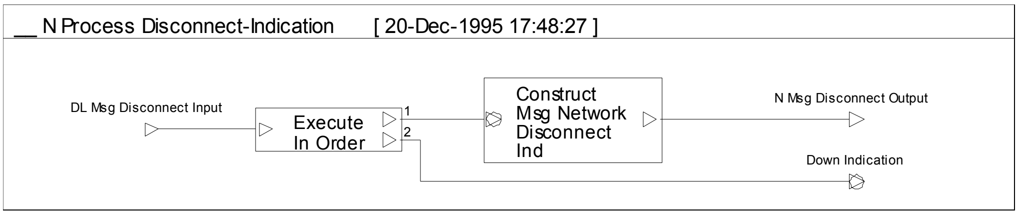
2.2.2.2. Process Connect Indication

This Module implements “PSPEC 1.3: Process Connect Message”. Note that “DFD 1: Process Datalink Message” has been subsumed by the Top.


__ N Process Connect-Indication [20-Dec-1995 17:47:56]
DL Msg Connect Input
Execute In Order
2
1
Construct Msg Network Connect Ind
N Msg Connect Output
Up Indication

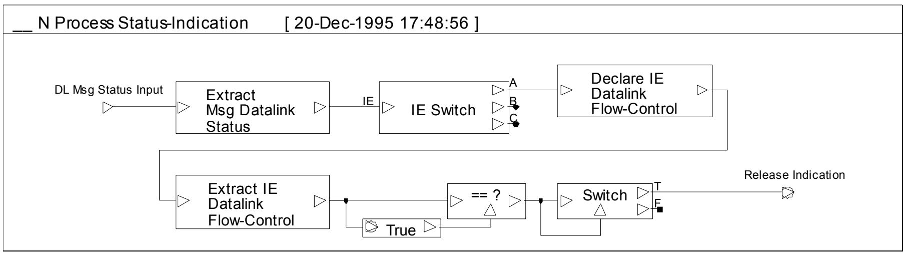
2.2.2.3. Process Disconnect Indication

This Module implements “PSPEC 1.4: Process Disconnect Message”. Note that “DFD 1: Process Datalink Message” has been subsumed by the Top.


__ N Process Disconnect-Indication [20-Dec-1995 17:48:27]
DL Msg Disconnect Input
Execute In Order
1
2
Construct Msg Network Disconnect Ind
N Msg Disconnect Output
Down Indication

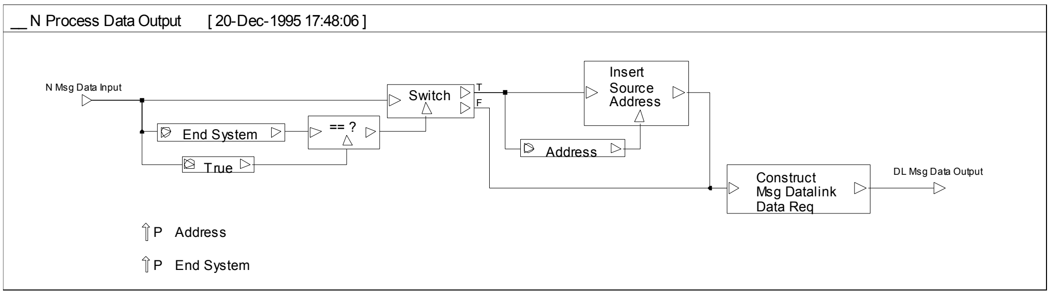
2.2.2.4. Process Status Indication

This Module implements “PSPEC 1.5: Process Status Message”. Note that “DFD 1: Process Datalink Message” has been subsumed by the Top.


__ N Process Status-Indication [20-Dec-1995 17:48:56]
DL Msg Status Input
Extract Msg Datalink Status
IE
IE Switch
A
B
C
Declare IE Datalink Flow-Control
Extract IE Datalink Flow-Control
True
== ?
Switch
T
F
Release Indication

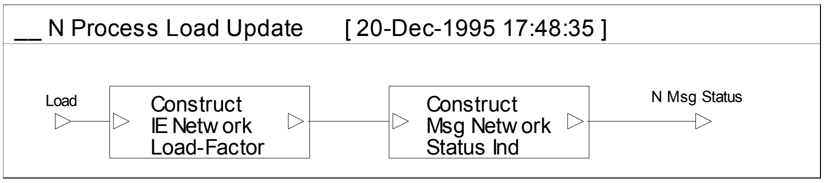
2.2.2.5. Process Data Output

This Module implements “DFD 2: Encapsulate for Datalink”.



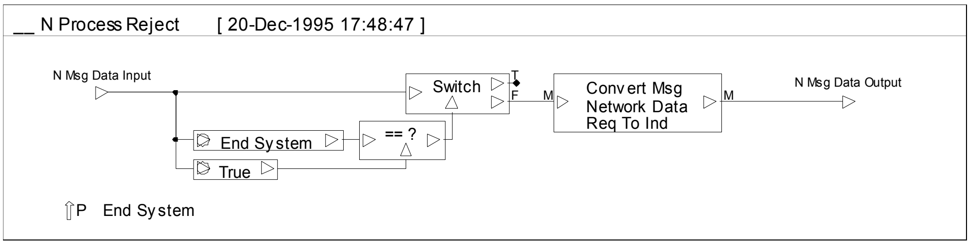
2.2.2.6. Process Load Update

This Module implements “PSPEC 4: Process Load Update”.


__ N Process Load Update [20-Dec-1995 17:48:35]
Load
Construct IE Network Load-Factor
Construct Msg Network Status Ind
N Msg Status

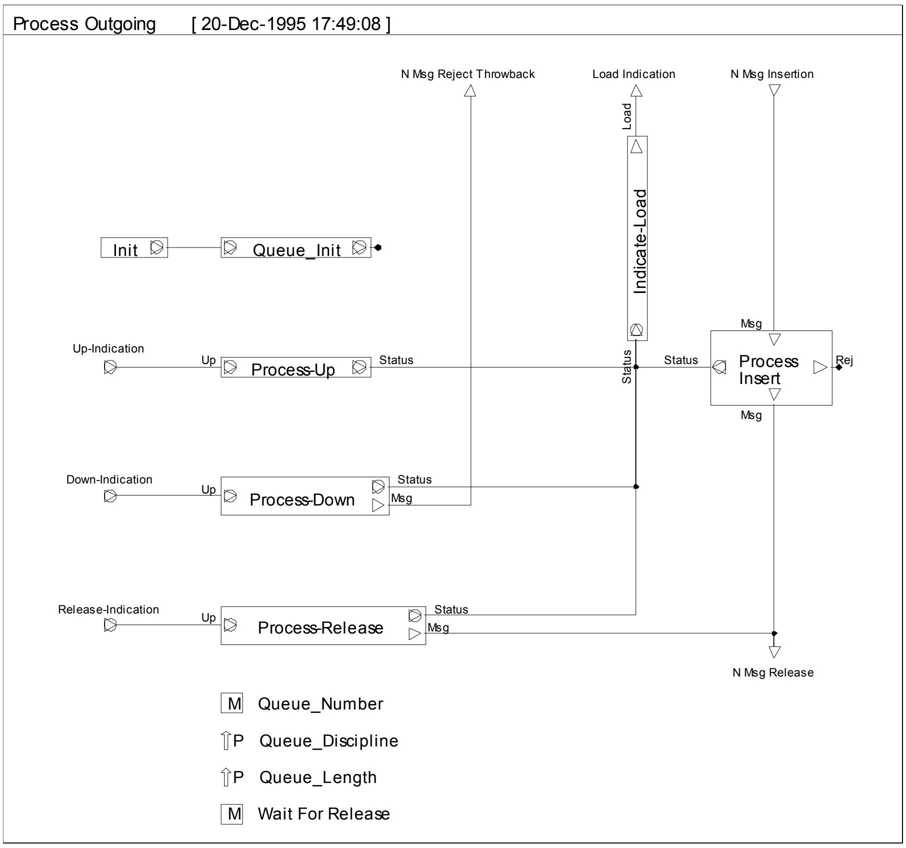
2.2.2.7. Process Reject

This Module implements “PSPEC 5: Process Reject Message”.


__ N Process Reject [20-Dec-1995 17:48:47]
N Msg Data Input
Switch
T
F
M
Convert Msg Network Data Req To Ind
M
N Msg Data Output
End System
== ?
True
P End System

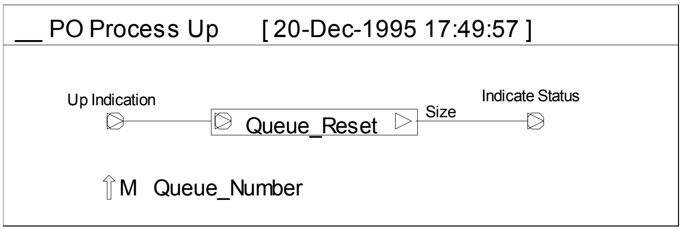
2.2.2.8. Process Outgoing

This Module implements “DFD 3: Process Outgoing Message”.



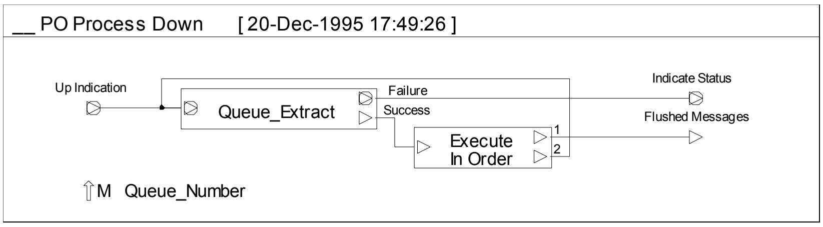
2.2.2.9. Process Outgoing -- Process Up

This Module implements “PSPEC 3.1: Initialise Queue”.



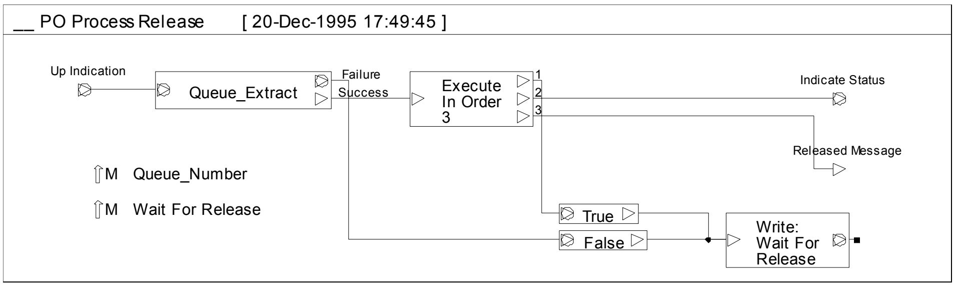
2.2.2.10. Process Outgoing -- Process Down

This Module implements “PSPEC 3.2: Flush Queue”.



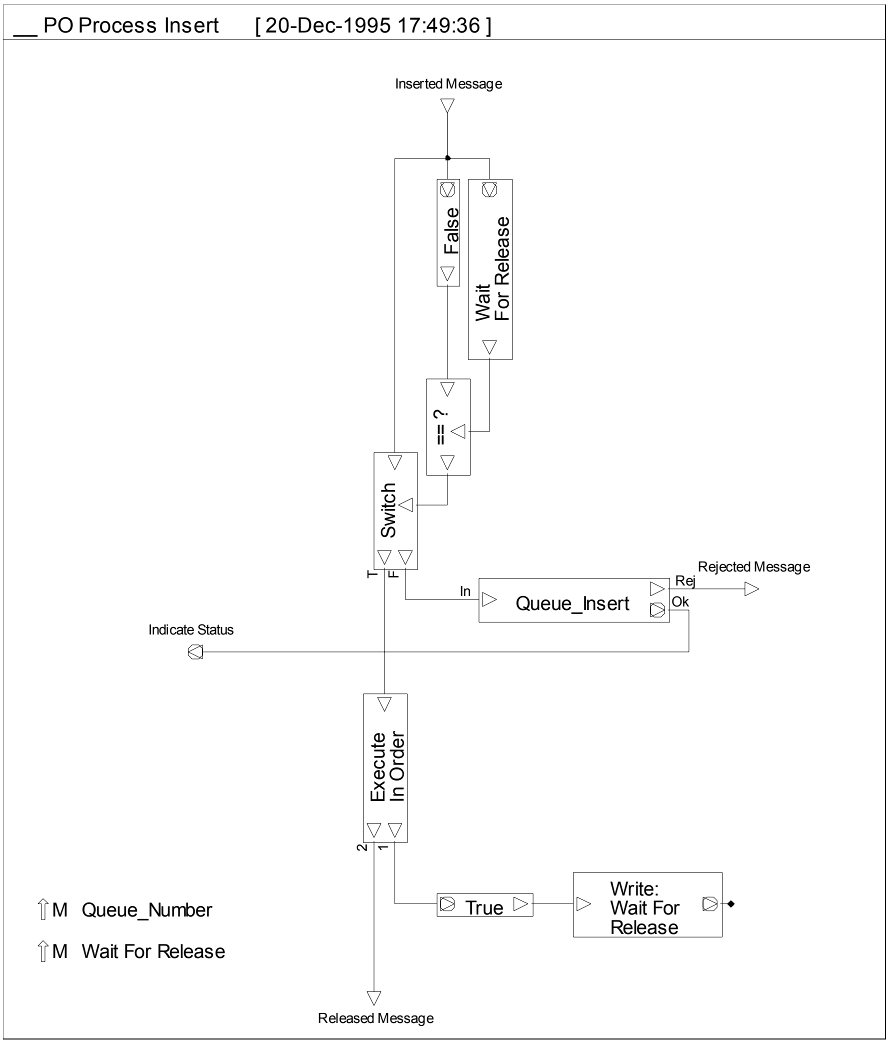
2.2.2.11. Process Outgoing -- Process Release

This Module implements “PSPEC 3.3: Release Queue”.



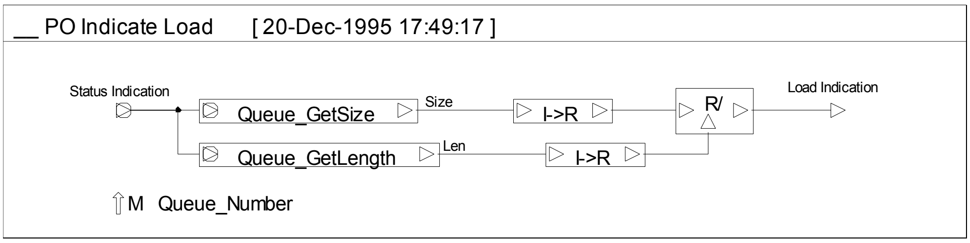
2.2.2.12. Process Outgoing -- Process Insert

This Module implements “PSPEC 3.4: Insert Queue”.



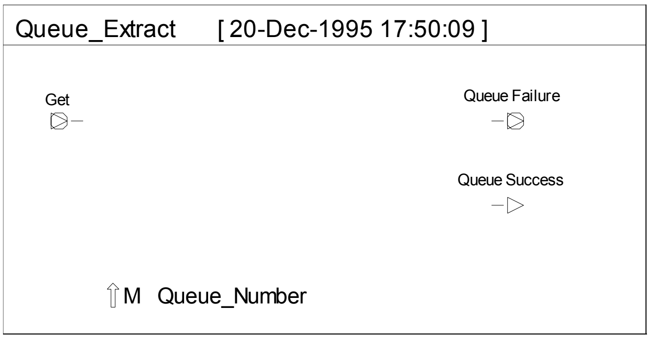
2.2.2.13. Process Outgoing -- Indicate Load

This Module implements “PSPEC 3.5: Indicate Load”.



2.2.2.14. Queue Extract

This Module acts as the interface betwen BONeS and the *Queue* ADT, as implemented in ‘C’, for the extraction of a single element from the Queue associated with the given *Queue_Number.*

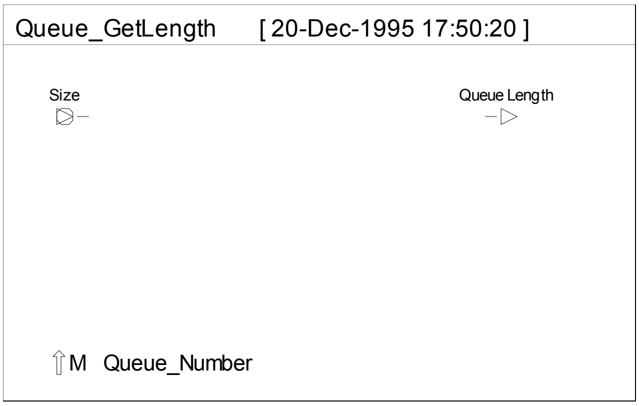


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* ------------------------------------------------------------------- */
#   include    "/u/mgream/BONeS/Constructed/Queue/BONeS_Queue_Extract.c"
/* ------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

   /* User RUN Below Here */

/* ------------------------------------------------------------------- */
   BONeS_Queue_Extract (Get, QueueSuccess, QueueFailure, argvector);
/* ------------------------------------------------------------------- */

   /* User RUN Above Here */

```

2.2.2.15. Queue Get Length

This Module acts as the interface betwen BONeS and the *Queue* ADT, as implemented in ‘C’, for the extraction of the length of the Queue associated with the given *Queue_Number.*

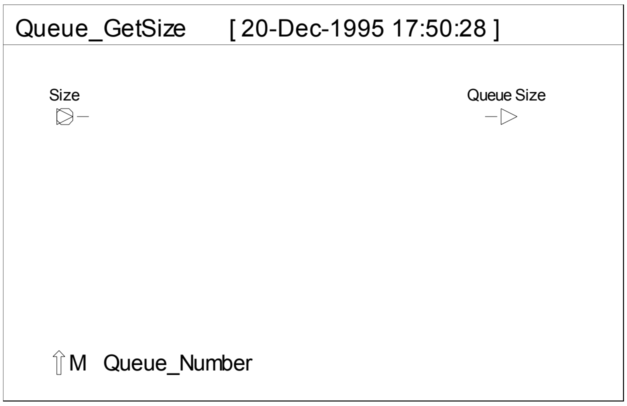


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* ------------------------------------------------------------------- */
#   include    "/u/mgream/BONeS/Constructed/Queue/BONeS_Queue_GetLength.c"
/* ------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

   /* User RUN Below Here */

/* ------------------------------------------------------------------- */
   BONeS_Queue_GetLength (Size, QueueLength, argvector);
/* ------------------------------------------------------------------- */

   /* User RUN Above Here */

```

2.2.2.16. Queue Get Size

This Module acts as the interface betwen BONeS and the *Queue* ADT, as implemented in ‘C’, for the extraction of the current size of the Queue associated with the given *Queue_Number.*



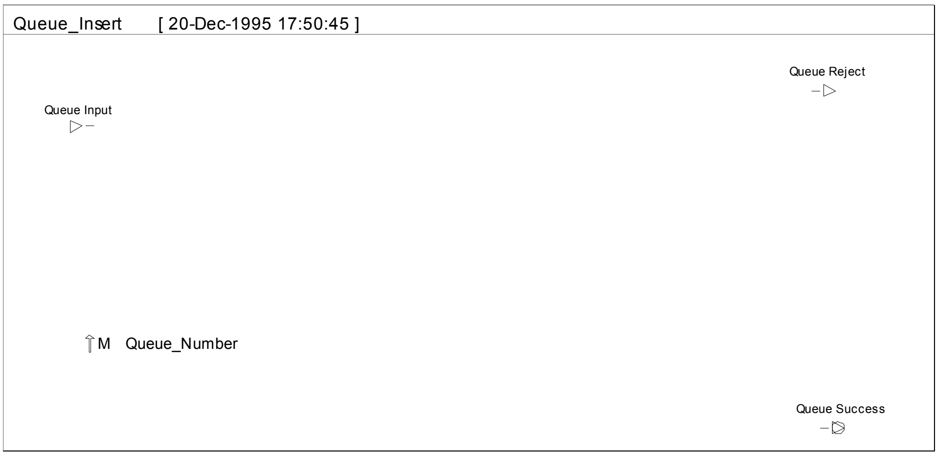
Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* ------------------------------------------------------------------- */
#   include    "/u/mgream/BONeS/Constructed/Queue/BONeS_Queue_GetSize.c"
/* ------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */
```

```

...

        /* User RUN Below Here */

/* --------------------------------------------------------------------- */
   BONeS_Queue_GetSize (Size, QueueSize, argvector);
/* --------------------------------------------------------------------- */

   /* User RUN Above Here */

```

2.2.2.17. Queue Insert

This Module acts as the interface betwen BONeS and the *Queue* ADT, as implemented in ‘C’, for the insertion of a single element to the Queue associated with the given *Queue_Number.*

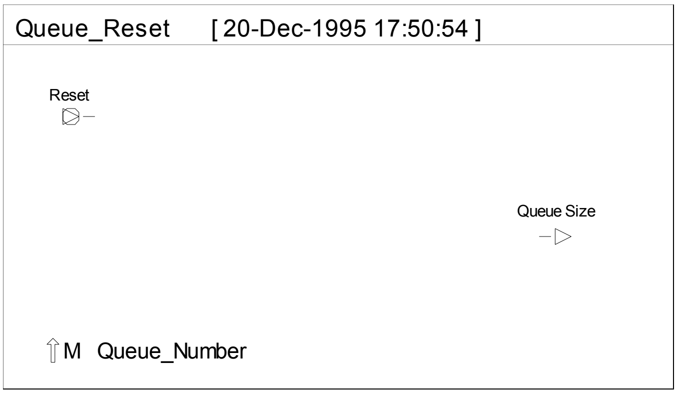


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* --------------------------------------------------------------------- */
#   include    "/u/mgream/BONeS/Constructed/Queue/BONeS_Queue_Insert.c"
/* --------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

        /* User RUN Below Here */

/* --------------------------------------------------------------------- */
   BONeS_Queue_Insert (QueueInput, QueueSuccess, QueueReject, argvector);
/* --------------------------------------------------------------------- */

   /* User RUN Above Here */

```

2.2.2.18. Queue Reset

This Module acts as the interface betwen BONeS and the *Queue* ADT, as implemented in ‘C’, to reset the Queue associated with the given *Queue_Number.*

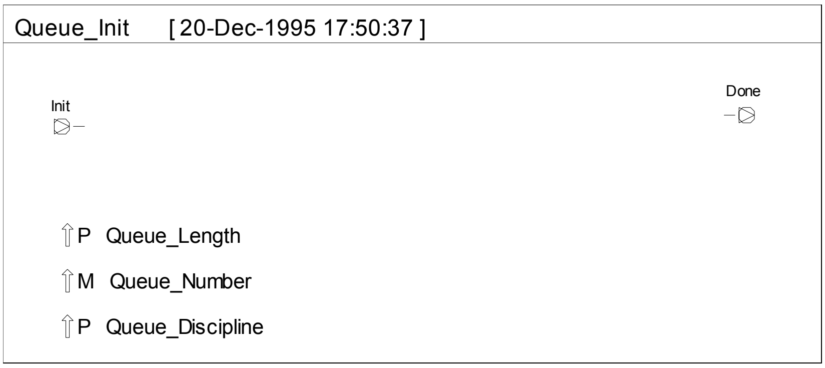


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* ------------------------------------------------------------------- */
#   include    "/u/mgream/BONeS/Constructed/Queue/BONeS_Queue_Reset.c"
/* ------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

   /* User RUN Below Here */

/* ------------------------------------------------------------------- */
    BONeS_Queue_Reset (Reset, QueueSize, argvector);
/* ------------------------------------------------------------------- */

   /* User RUN Above Here */

```

2.2.2.19. Queue Init

This Module acts as the interface betwen BONeS and the *Queue* ADT, as implemented in ‘C’, to initialise the Queue with the given *Queue_Length* and *Queue_Discpline*, to provide a *Queue_Number.*



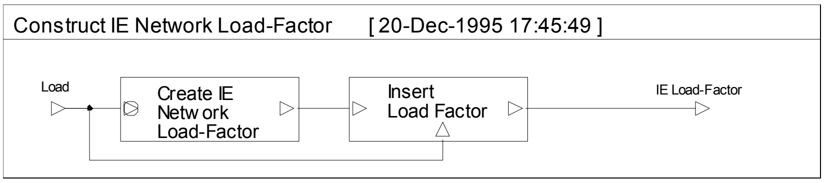
Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* ------------------------------------------------------------------- */
#    include      "/u/mgream/BONeS/Constructed/Queue/BONeS_Queue_Create.c"
/* ------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

```

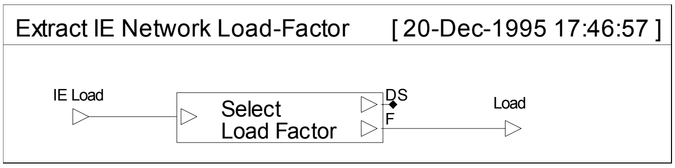
```
        /* User RUN Below Here */

/* -------------------------------------------------------------------- */

    BONeS_Queue_Create (argvector);
    __FreeArc (Init);
    __GenerateTrigger (Done);

/* -------------------------------------------------------------------- */

        /* User RUN Above Here */

```

2.2.3. Support Modules

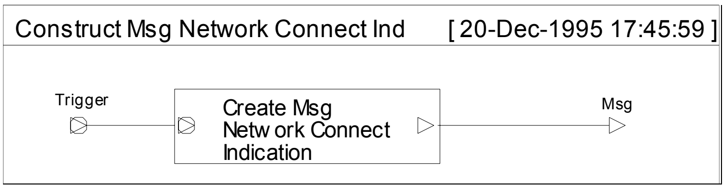
2.2.3.1. Construct IE Network Load Factor


Construct IE Network Load-Factor [20-Dec-1995 17:45:49]
Load
Create IE Network Load-Factor
Insert Load Factor
IE Load-Factor

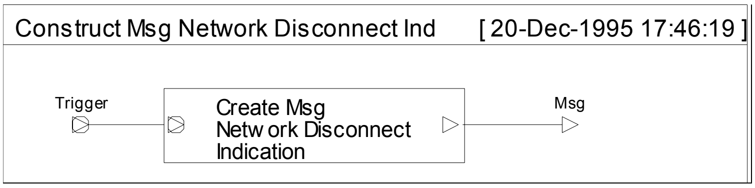
2.2.3.2. Extract IE Network Load Factor



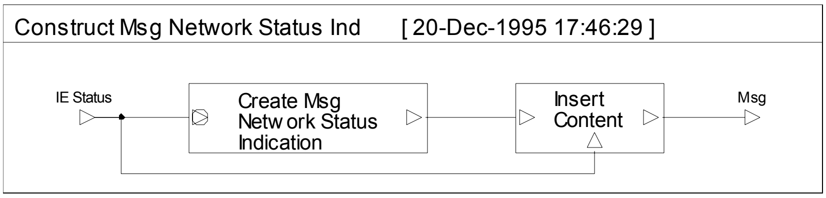
2.2.3.3. Construct Msg Network Connect Indication


Construct Msg Network Connect Ind [20-Dec-1995 17:45:59]
Trigger
Create Msg Network Connect Indication
Msg

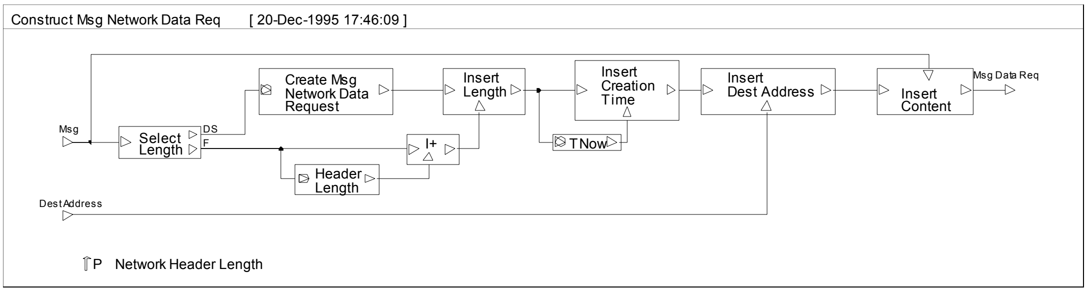
2.2.3.4. Construct Msg Network Disconnect Indication


Construct Msg Network Disconnect Ind [20-Dec-1995 17:46:19]
Trigger
Create Msg Network Disconnect Indication
Msg

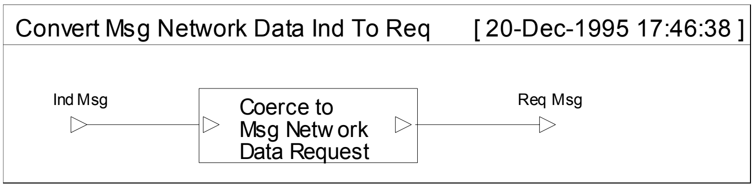
2.2.3.5. Construct Msg Network Status Indication


Construct Msg Network Status Ind [20-Dec-1995 17:46:29]
IE Status
Create Msg Network Status Indication
Insert Content
Msg

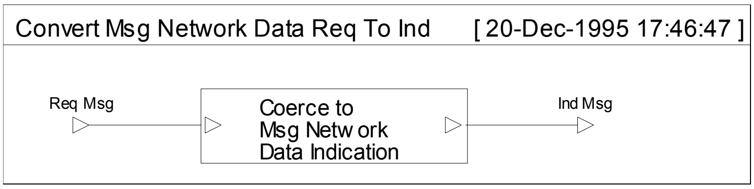
2.2.3.6. Construct Msg Network Data Request



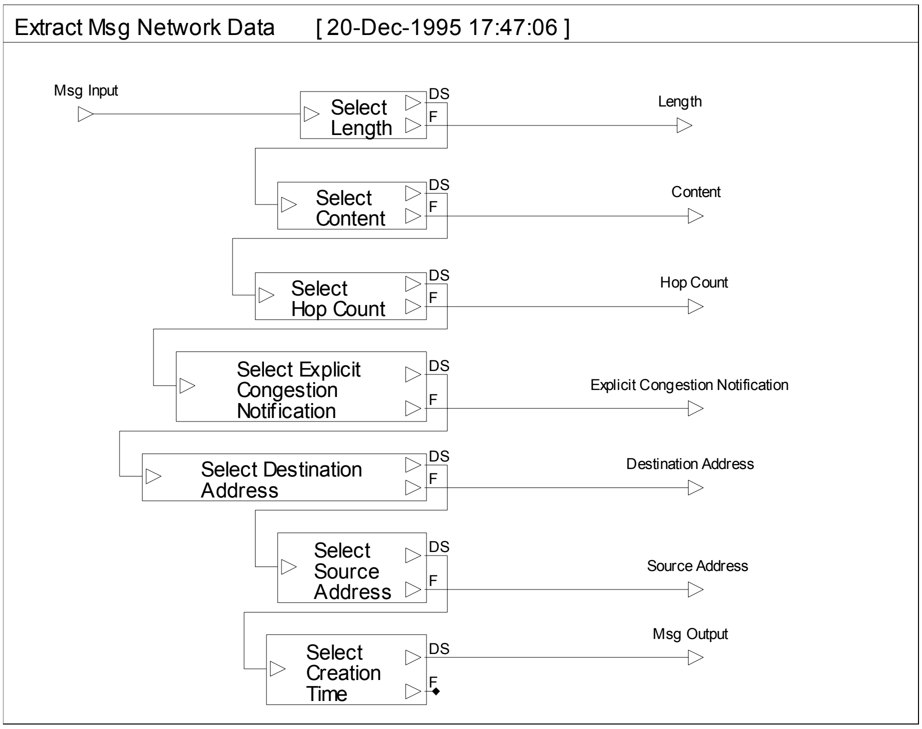
2.2.3.7. Convert Msg Network Data Indication to Request


Convert Msg Network Data Ind To Req [20-Dec-1995 17:46:38]
Ind Msg
Coerce to Msg Network Data Request
Req Msg

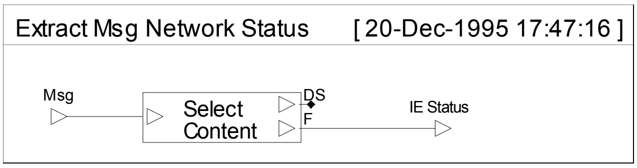
2.2.3.8. Convert Msg Network Data Request to Indication


Convert Msg Network Data Req To Ind [20-Dec-1995 17:46:47]
Req Msg
Coerce to Msg Network Data Indication
Ind Msg

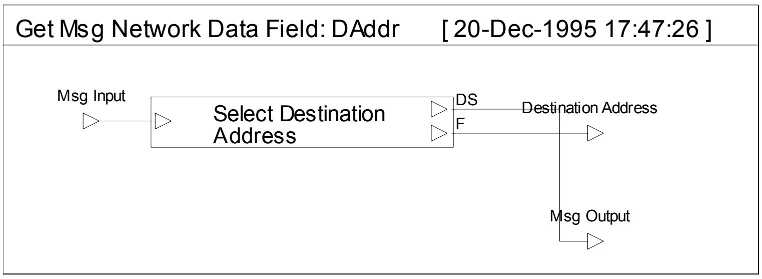
2.2.3.9. Extract Msg Network Data



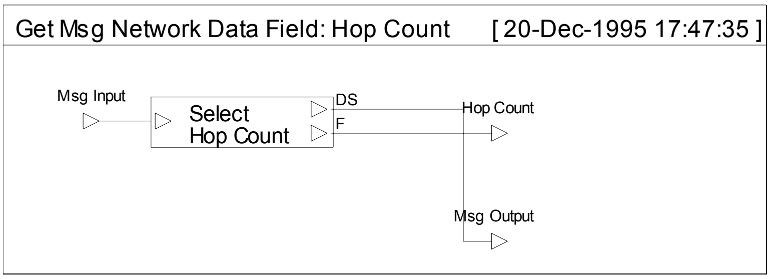
2.2.3.10. Extract Msg Network Status



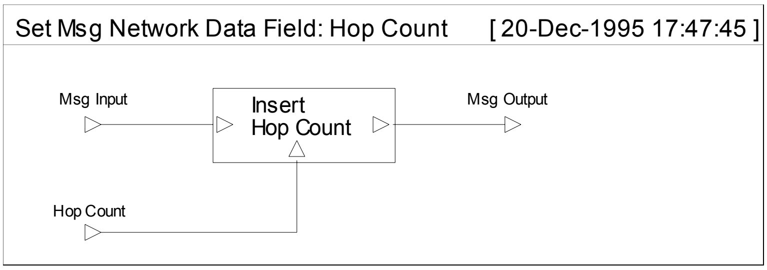
2.2.3.11. Get Msg Network Data Field : Destination Address


Get Msg Network Data Field: DAddr [20-Dec-1995 17:47:26]
Msg Input
Select Destination Address
DS
F
Destination Address
Msg Output

2.2.3.12. Get Msg Network Data Field : Hop Count


Get Msg Network Data Field: Hop Count [20-Dec-1995 17:47:35]
Msg Input
Select Hop Count
DS
F
Hop Count
Msg Output

2.2.3.13. Set Msg Network Data Field : Hop Count


Set Msg Network Data Field: Hop Count [20-Dec-1995 17:47:45]
Msg Input
Insert Hop Count
Msg Output
Hop Count

2.2.4. ‘C’ Modules

The ‘C’ Modules for the Queue consist of top level interface functions that use lower level Queue Primitives and a Queue Table.

2.2.4.1. BONeS Queue Create (Init)

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_Queue_Create.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: BONeS_Queue_Create.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   define      DECLARE_MAIN_VARIABLES
#   include     "/u/mgream/BONeS/Constructed/Queue/Queue.c"
/* -------------------------------------------------------------------- */
/*  ParseTable
 |
 |  The ParseEntry and ParseTable are used to contain items that can
 |  be specified as options for the queue.
 */
typedef struct _ParseEntry_ST
  {
    char * Token;
    int Options;
  } _ParseEntry;

static _ParseEntry _ParseTable[] =
  {
    { "droptail",       QUEUE_OPT_DROPTAIL },
    { "droprandom",     QUEUE_OPT_DROPRANDOM },
    { "red",            QUEUE_OPT_RED },
    { "priosize",       QUEUE_OPT_PRIOSIZE },
    { "prioclass",      QUEUE_OPT_PRIOCLASS },
    { "addrqueue",      QUEUE_OPT_ADDRESS },
    { "default",        QUEUE_OPT_DEFAULT },
  };

#define _PARSE_TABLE_SZ (sizeof (_ParseTable) / sizeof (_ParseEntry))

/* -------------------------------------------------------------------- */

/*  _toupper
 |
 |  covert the passed character to upper case.
 */
static char _toupper (ch)
    char ch;
  {
    return (ch >= 'a' && ch <= 'z') ? (ch - 'a' + 'A') : ch;
  }

/* -------------------------------------------------------------------- */

/*  _isspace
 |
 |  is the passed character a whitespace ?
 */
static char _isspace (ch)
    char ch;
  {
    return (ch == ' ' || ch == '\t' || ch == ',' || ch == ':' || ch == ';');
  }

/* -------------------------------------------------------------------- */

/*  _strcasecmp
 |
 |  string compare without considering case
 */
static int _strcasecmp (StringA, StringB)
    char * StringA;
    char * StringB;
  {
    while (_toupper (*StringA) == _toupper (*StringB))
      {
        if (*StringA == '\0')
            return 0;

        StringA++; StringB++;
      }
    return 1;
  }

/* -------------------------------------------------------------------- */

/*  _ParseOptions
 |
 |  Using the given table of option keywords, attempt to decompose
 |  a given string into a set of flags; noting that there is no
 |  semantic check here at all, it's all syntactic.
 */
static int _ParseOptions (String)
    char * String;
  {
    char * Token;
    int Options = 0;
    int Index;

    while (*String != '\0')
      {
        /* SKIP WHITESPACE */
        while (_isspace (*String) && *String != '\0')
            String++;
        if (*String == '\0')
            break;

        /* EXTRACT TOKEN */
        Token = String;
        while (!_isspace (*String) && *String != '\0')
            String++;
        if (*String != '\0')
            *String++ = '\0';

        /* PARSE TOKEN */
        for (Index = 0; Index < _PARSE_TABLE_SZ &&
             _strcasecmp (Token, _ParseTable[Index].Token) != 0; Index++)
            ;

        if (Index < _PARSE_TABLE_SZ)
            Options |= _ParseTable[Index].Options;
      }

    return (Options == 0) ? QUEUE_OPT_DEFAULT : Options;
  }

/* -------------------------------------------------------------------- */

/*  BONeS_Queue_Create
 |
 |  This is where we instantiate the queue for a specific use; what
 |  occurs is that a string that has the discipline options is parsed
 |  to determine the flags that will be used with this queue. Then we
 |  allocate an entry in the table and set up the appropriate mapping.
 */
static void BONeS_Queue_Create (argvector)
    arg_ptr argvector;
  {
    char * OptionsString = __GetSTRINGVal (Queue_Discipline_arc);
    int _Options = _ParseOptions (OptionsString);
    int _Length = __GetINTEGERVal (Queue_Length_arc);
    int QIndex = QueueTableAlloc (_Length, _Options);

    if (QIndex < 0 || QIndex >= QUEUE_TABLE_SZ)
      {
        __ReportError (MODULE_NAMESTRING, "Queue Alloc failed!");
        QIndex = QUEUE_TABLE_SZ;
      }

    __PutINTEGERVal (Queue_Number_arc, QIndex);
    __Bfree (OptionsString);
  }

/* -------------------------------------------------------------------- */

```

2.2.4.2. BONeS Queue Destroy (Init)

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_Queue_Destroy.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: BONeS_Queue_Destroy.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Queue/Queue.c"
/* -------------------------------------------------------------------- */

/*  _BONeS_Queue_Destroy
 |
 |  Destroy the queue by removing its mapping and invalidating the
 |  index that we maintain for it.
 */
static void BONeS_Queue_Destroy (argvector)
    arg_ptr argvector;
  {
    int QIndex = __GetINTEGERVal (Queue_Number_arc);
    QueueEntry * QEntry = &QueueTable[QIndex];

    if (QIndex < QUEUE_TABLE_SZ && QEntry->Allocated == TRUE)
      {
        QueueTableFree (QIndex);
      }

    __PutINTEGERVal (Queue_Number_arc, QUEUE_TABLE_SZ);
  }

/* -------------------------------------------------------------------- */

```

2.2.4.3. BONeS Queue Extract

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_Queue_Extract.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: BONeS_Queue_Extract.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Queue/Queue.c"
/* -------------------------------------------------------------------- */

#   define  SIZE_THRESHOLD  128

/* -------------------------------------------------------------------- */

static  int                 _fh_Initialised = 0;
static  field_handle        _fh_Length;
static  field_handle        _fh_SourceAddress;
static  field_handle        _fh_DestAddress;
#ifdef CLASS_PRIORITY
static  field_handle        _fh_Class;
#endif
static  type_handle         _th_Msg_Network_Data;

/* -------------------------------------------------------------------- */

/*  _fh_Initialise
 |
 |  Initialise the Field and Type handles for use with this module.
 */
static void _fh_Initialise ()
  {
    if (_fh_Initialised != 0)
        return;
    _fh_Initialised = 1;
    _th_Msg_Network_Data = __GetTypeHandleId (MsgNetworkDataRequest);
    _fh_Length = __GetFldHandleId (_th_Msg_Network_Data, Field_Length);
    _fh_SourceAddress = __GetFldHandleId (_th_Msg_Network_Data,
                Field_SourceAddress);
    _fh_DestAddress = __GetFldHandleId (_th_Msg_Network_Data,
                Field_DestAddress);
#ifdef CLASS_PRIORITY
    _fh_Class = __GetFldHandleId (_th_Msg_Network_Data, Field_Class);
#endif
  }

/* -------------------------------------------------------------------- */

/*  _Get_Src_Address
```

```
 |
 |  Return the address (Source Address) of a BONeS message.
 */
static int _Get_Src_Address (Msg)
    arc_ptr Msg;
  {
    if (_fh_Initialised == 0)
        _fh_Initialise ();
    return __GetINTEGERFldVal (Msg, _fh_SourceAddress);
  }

/* -------------------------------------------------------------------- */

/*  _Get_Dst_Address
 |
 |  Return the address (Dest Address) of a BONeS message.
 */
static int _Get_Dst_Address (Msg)
    arc_ptr Msg;
  {
    if (_fh_Initialised == 0)
        _fh_Initialise ();
    return __GetINTEGERFldVal (Msg, _fh_DestAddress);
  }

/* -------------------------------------------------------------------- */

/*  _Get_Size
 |
 |  Return the size (Length) of a BONeS message.
 */
static int _Get_Size (Msg)
    arc_ptr Msg;
  {
    if (_fh_Initialised == 0)
        _fh_Initialise ();
    return __GetINTEGERFldVal (Msg, _fh_Length);
  }

/* -------------------------------------------------------------------- */

#ifdef CLASS_PRIORITY
/*  _Get_Class
 |
 |  Return the class (Class) of a BONeS message.
 */
static int _Get_Class (Msg)
    arc_ptr Msg;
  {
    if (_fh_Initialised == 0)
        _fh_Initialise ();
    return __GetINTEGERFldVal (Msg, _fh_Class);
  }
#endif

/* -------------------------------------------------------------------- */

/*  _Filter_On_Address
 |
 |  Attempt to extract a message with the next address after the
 |  previous message. What we do is extract a message, and store
 |  its address, then on the next time around, we try for the one
 |  following this. The strategy is thus: iterate through the
 |  queue and examine the addresses to find either one greater, or
 |  the lowest one. The filter then turns off any that are not
 |  applicable.
 */
static void _Filter_On_Address (QEntry, FilterArray, FilterLength)
    QueueEntry * QEntry;
    int * FilterArray;
    int FilterLength;
  {
    int Index;
    int Address;
    int CmpAddress;
    int MinAddress;
    int Size;

    Size = QueueSize (QEntry->Que);
    CmpAddress = -1;
    MinAddress = -1;

    for (Index = 0; Index < Size && Index < FilterLength; Index++)
      {
        Address = _Get_Src_Address (QueuePeekElement (QEntry->Que, Index));

        if (CmpAddress == -1 ||
            (Address < CmpAddress && CmpAddress > QEntry->Ext_AddressLast))
            CmpAddress = Address;

        if (MinAddress == -1 || Address < MinAddress)
            MinAddress = Address;
      }

    Address = (CmpAddress == -1) ? MinAddress : CmpAddress;

    for (Index = 0; Index < Size && Index < FilterLength; Index++)
      {
        if (_Get_Src_Address (QueuePeekElement (QEntry->Que, Index)) != Address)
          {
            FilterArray[Index] = FALSE;
          }
      }
  }

/* ------------------------------------------------------------------- */

/*  _Filter_On_Size
 |
 |  This filter attempts to alternatively extract Short and Long
 |  packets the other side of a specified threshold; the purpose
 |  mainly is to allow interactive packets to have a slight priority
 |  (and ack packets as well!!) over bulk data packets. We do need
 |  to alternate otherwise we could starve the big packets. The
 |  strategy is thus:
 |      Iterate through all the entirse and look at the size of
 |      each entry, if the size is in the same threshold direction
 |      as the last entry, then do remove that entry.
 |  There are still some small questions about this policy, i.e.
 |  it is not entirely fair all the time ...
 |
 */
static void _Filter_On_Size (QEntry, FilterArray, FilterLength)
    QueueEntry * QEntry;
    int * FilterArray;
    int FilterLength;
  {
    int Index;
    int Size;

    Size = QueueSize (QEntry->Que);
    for (Index = 0; Index < Size && Index < FilterLength; Index++)
      {
        int Length = _Get_Size (QueuePeekElement (QEntry->Que, Index));

        if (QEntry->Ext_SizeLast < SIZE_THRESHOLD && Length < SIZE_THRESHOLD)
          {
            FilterArray[Index] = FALSE;
          }
        else if (QEntry->Ext_SizeLast >= SIZE_THRESHOLD &&
                    Length >= SIZE_THRESHOLD)
          {
            FilterArray[Index] = FALSE;
          }

      }
  }

/* ------------------------------------------------------------------- */

/*  _Filter_On_Class
 |
 |  Not implemented.
 */
static void _Filter_On_Class (QEntry, FilterArray, FilterLength)
    QueueEntry * QEntry;
    int * FilterArray;
    int FilterLength;
  {
    /* Nothing */
  }

```

```
/* ------------------------------------------------------------------ */

/*  _Extract_Msg
 |
 |  To support the different extraction mechanisms, what we do is set
 |  up an array to hold a flag for each entry that there is in the
 |  queue, and then we go off and filter through all the possible
 |  options, only keeping the flags turned on for those that satisfy
 |  each one. At the end, we select the first one that is available,
 |  or just go straight for the head if we have an otherwise nomatch.
 */
static void _Extract_Msg (QEntry, Success, Failure)
    QueueEntry * QEntry;
    arc_ptr Success;
    arc_ptr Failure;
  {
    char * FilterArray;
    int FilterLength;
    int Index;

    FilterLength = QueueLength (QEntry->Que);
    FilterArray = (char *) __Balloc (sizeof (char) * FilterLength, "FArray");
    for (Index = 0; Index < FilterLength; Index++)
        FilterArray[Index] = TRUE;

    if (QEntry->Options & QUEUE_OPT_ADDRESS)
        _Filter_On_Address (QEntry, FilterArray, FilterLength);

    if (QEntry->Options & QUEUE_OPT_PRIOSIZE)
        _Filter_On_Size (QEntry, FilterArray, FilterLength);

    if (QEntry->Options & QUEUE_OPT_PRIOCLASS)
        _Filter_On_Class (QEntry, FilterArray, FilterLength);

    for (Index = 0; Index < FilterLength && FilterArray[Index]== FALSE; Index++)
        ;

    if (Index < FilterLength)
      {
        arc_ptr arc = QueueGetElement (QEntry->Que, Index);
        __CopyArc (arc, Success);
        __FreeArc (arc);
        __Bfree ((char *)arc);
      }
    else
      {
        arc_ptr arc = QueueGetHead (QEntry->Que);
        __CopyArc (arc, Success);
        __FreeArc (arc);
        __Bfree ((char *)arc);
      }

    QEntry->Ext_AddressLast = _Get_Address (Success);
    QEntry->Ext_SizeLast = _Get_Size (Success);

    __Bfree (FilterArray);
  }

/* ------------------------------------------------------------------ */

/*  BONeS_Queue_Extract
 |
 |  Extract the next entry from the Queue using whatever discipline
 |  we have specified. This amounts to first ensuring that we do
 |  have something in the queue as a precondition to carrying out
 |  the extraction.
 */
static void BONeS_Queue_Extract (InTrigger, Success, Failure, argvector)
    arc_ptr InTrigger;
    arc_ptr Success;
    arc_ptr Failure;
    arg_ptr argvector;
  {
    int QIndex = __GetINTEGERVal (Queue_Number_arc);
    QueueEntry * QEntry = &QueueTable[QIndex];

    if (QIndex < QUEUE_TABLE_SZ && QEntry->Allocated == TRUE)
      {
        if (QueueSize (QEntry->Que) == 0)
          {
            __GenerateTrigger (Failure);
          }
        else
          {
            _Extract_Msg (QEntry, Success, Failure);
          }
      }

    __FreeArc (InTrigger);
  }

/* --------------------------------------------------------------------- */

```

2.2.4.4. BONeS Queue Insert

```

/* --------------------------------------------------------------------- */
/* $Id: BONeS_Queue_Insert.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: BONeS_Queue_Insert.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* --------------------------------------------------------------------- */

/* --------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Queue/Queue.c"
/* --------------------------------------------------------------------- */

/*  _Arc_Clone
 |
 |  Make a copy of the passed arc, and return it.
 */
static arc_ptr _Arc_Clone (arc)
    arc_ptr arc;
  {
    arc_ptr clone = (arc_ptr) __Balloc (sizeof (arc_t), "Arc");
    clone->enable = 0;
    __CopyArc (arc, clone);
    return clone;
  }

/* --------------------------------------------------------------------- */

/*  _Insert_DropTail
 |
 |  Insert a message into the queue using a DROP TAIL policy; this
 |  amounts to simply dropping the input message if the queue is
 |  already full.
 */
static void _Insert_DropTail (QEntry, Msg, Success, Failure)
    QueueEntry * QEntry;
    arc_ptr Msg;
    arc_ptr Success;
    arc_ptr Failure;
  {
    if (QueueSize (QEntry->Que) < QueueLength (QEntry->Que))
      {
        QueueInsert (QEntry->Que, _Arc_Clone (Msg));
        __GenerateTrigger (Success);
      }
    else
      {
        __CopyArc (Msg, Failure);
      }
  }

/* --------------------------------------------------------------------- */

/*  _Insert_DropRandom
 |
 |  Insert a message into the queue using the DROP RANDOM policy; this
 |  amounts to dropping a random entry if the queue is already full,
 |  and then placing the input message onto the end of the queue. Note
 |  that both FAILURE and SUCCESS outputs can be enabled.
 */
```

```
static void _Insert_DropRandom (QEntry, Msg, Success, Failure)
    QueueEntry * QEntry;
    arc_ptr Msg;
    arc_ptr Success;
    arc_ptr Failure;
  {
    if (QueueSize (QEntry->Que) >= QueueLength (QEntry->Que))
      {
        QueueIndex QOffset = _UNIFORM (0, QueueSize (QEntry->Que));
        arc_ptr arc = QueueGetElement (QEntry->Que, QOffset);

        __CopyArc (arc, Failure);
        __FreeArc (arc);
        __Bfree ((char *)arc);
      }

    QueueInsert (QEntry->Que, _Arc_Clone (Msg));
    __GenerateTrigger (Success);
  }

/* -------------------------------------------------------------------- */

/*  _Insert_RED
 |
 */
static void _Insert_RED (QEntry, Msg, Success, Failure)
    QueueEntry * QEntry;
    arc_ptr Msg;
    arc_ptr Success;
    arc_ptr Failure;
  {
    __CopyArc (Msg, Failure);
  }

/* -------------------------------------------------------------------- */

/*  BONeS_Queue_Insert
 |
 |  Insert a message into the queue, what we do is locate the specific
 |  policy that is being used and then ask it to carry out the
 |  insertion.
 */
static void BONeS_Queue_Insert (Msg, Success, Failure, argvector)
    arc_ptr Msg;
    arc_ptr Success;
    arc_ptr Failure;
    arg_ptr argvector;
  {
    int QIndex = __GetINTEGERVal (Queue_Number_arc);
    QueueEntry * QEntry = &QueueTable[QIndex];

    if (QIndex < QUEUE_TABLE_SZ && QEntry->Allocated == TRUE)
      {
        if (QEntry->Options & QUEUE_OPT_DROPTAIL)
          {
            _Insert_DropTail (QEntry, Msg, Success, Failure);
          }
        else if (QEntry->Options & QUEUE_OPT_DROPRANDOM)
          {
            _Insert_DropRandom (QEntry, Msg, Success, Failure);
          }
        else if (QEntry->Options & QUEUE_OPT_RED)
          {
            _Insert_RED (QEntry, Msg, Success, Failure);
          }
        else
          {
            __CopyArc (Msg, Failure);
          }
      }

    __FreeArc (Msg);
  }

/* -------------------------------------------------------------------- */

```

2.2.4.5. BONeS Queue Get Length

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_Queue_GetLength.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: BONeS_Queue_GetLength.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Queue/Queue.c"
/* -------------------------------------------------------------------- */

/*  BONeS_Queue_GetLength
 |
 |  Get the length of a specific queue; i.e. the fixed length, not
 |  the number of elements that are contained in the queue at a
 |  particular time.
 */
static void BONeS_Queue_GetLength (InTrigger, Length, argvector)
    arc_ptr InTrigger;
    arc_ptr Length;
    arg_ptr argvector;
  {
    int QIndex = __GetINTEGERVal (Queue_Number_arc);
    QueueEntry * QEntry = &QueueTable[QIndex];

    if (QIndex < QUEUE_TABLE_SZ && QEntry->Allocated == TRUE)
      {
        __PutINTEGERVal (Length, QueueLength (QEntry->Que));
      }

    __FreeArc (InTrigger);
  }

/* -------------------------------------------------------------------- */

```

2.2.4.6. BONeS Queue Get Size

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_Queue_GetSize.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: BONeS_Queue_GetSize.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Queue/Queue.c"
/* -------------------------------------------------------------------- */

/*  BONeS_Queue_GetSize
 |
 |  Get the size of a queue; i.e. the current number of elements
 |  that are contained within the queue.
 */
static void BONeS_Queue_GetSize (InTrigger, Size, argvector)
    arc_ptr InTrigger;
    arc_ptr Size;
    arg_ptr argvector;
  {
    int QIndex = __GetINTEGERVal (Queue_Number_arc);
    QueueEntry * QEntry = &QueueTable[QIndex];

    if (QIndex < QUEUE_TABLE_SZ && QEntry->Allocated == TRUE)
      {
        __PutINTEGERVal (Size, QueueSize (QEntry->Que));
      }

    __FreeArc (InTrigger);
  }
```

```

/* -------------------------------------------------------------------- */

```

2.2.4.7. BONeS Queue Reset

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_Queue_Reset.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: BONeS_Queue_Reset.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Queue/Queue.c"
/* -------------------------------------------------------------------- */

/*  BONeS_Queue_Reset
 |
 |  Reset the queue by killing all contents; we extract each item
 |  and free then kill it.
 */
static void BONeS_Queue_Reset (InTrigger, OutSize, argvector)
    arc_ptr InTrigger;
    arc_ptr OutSize;
    arg_ptr argvector;
  {
    int QIndex = __GetINTEGERVal (Queue_Number_arc);
    QueueEntry * QEntry = &QueueTable[QIndex];

    if (QIndex < QUEUE_TABLE_SZ && QEntry->Allocated == TRUE)
      {
        arc_ptr arc;

        while ((arc = QueueGetHead (QEntry->Que)) != NULL)
          {
            __FreeArc (arc);
            __Bfree ((char *)arc);
          }
      }

    __FreeArc (InTrigger);
    __PutINTEGERVal (OutSize, 0);
  }

/* -------------------------------------------------------------------- */

```

2.2.4.8. BONeS Queue (Primitive)

```

/* -------------------------------------------------------------------- */
/* $Id: Queue.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: Queue.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Queue/q_primitives.c"
#   include     "/u/mgream/BONeS/Constructed/Queue/q_table.c"
/* -------------------------------------------------------------------- */

```

2.2.4.8.1. Primitives

```c

/* -------------------------------------------------------------------- */
/* $Id: q_primitives.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: q_primitives.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

#ifdef TEST

#   include <stdio.h>
#   include <stdlib.h>
#   include <string.h>
#   include <assert.h>

#   define  _MALLOC(s,id)       malloc (s)
#   define  _FREE(p)            free (p)

#   define  _HANDLE             char *
#   define  _HANDLE_DESTROY(x)  if ((x) != NULL) { _FREE (x); (x) = NULL; }

#else

#   define  assert(x)
#   define  _MALLOC(s,id)       __Balloc (s, id)
#   define  _FREE(p)            __Bfree ((char *)p)

#   define  _HANDLE             arc_ptr
#   define  _HANDLE_DESTROY(x)  if ((x) != NULL) \
                                  { \
                                    __FreeArc (x); \
                                    __Bfree ((char*)x); \
                                    (x) = NULL; \
                                  }
#endif

/* -------------------------------------------------------------------- */

typedef int Boolean;

#   define  FALSE               (0)
#   define  TRUE                (1)

/* -------------------------------------------------------------------- */

typedef struct QueueItem_ST
  {
    struct QueueItem_ST * Next;
    Boolean Allocated;
    _HANDLE Handle;
  } QueueItem;

/* -------------------------------------------------------------------- */

typedef int QueueIndex;

/* -------------------------------------------------------------------- */

typedef struct Queue_ST
  {
    QueueIndex Length;
    QueueItem * _Pool;
    QueueItem * Head;
  } Queue;

/* -------------------------------------------------------------------- */

#ifndef P
#   ifdef ANSIC
#       define  P(x)            x
#   else
#       define  P(x)            ()
#   endif
#endif

```

```
static QueueItem * QueuePoolCapture P ((Queue * Que));
static void QueuePoolRelease P ((Queue * Que, QueueItem * QItem));
static Queue * QueueCreate P ((QueueIndex Length));
static void QueueDestroy P ((Queue * Que));
static QueueIndex QueueLength P ((Queue * Que));
static QueueIndex QueueSize P ((Queue * Que));
static _HANDLE QueueGetHead P ((Queue * Que));
static _HANDLE QueueGetElement P ((Queue * Que, QueueIndex QOffset));
static _HANDLE QueuePeekElement P ((Queue * Que, QueueIndex QOffset));
static Boolean QueueInsert P ((Queue * Que, _HANDLE Handle));

/* ------------------------------------------------------------------- */

static QueueItem * QueuePoolCapture (Que)
    Queue * Que;
  {
    QueueIndex QIndex;

    for (QIndex = 0; QIndex < Que->Length; QIndex++)
      {
        if (Que->_Pool[QIndex].Allocated == FALSE)
          {
            QueueItem * QItem = &Que->_Pool[QIndex];

            QItem->Allocated = TRUE;
            QItem->Next = NULL;
            QItem->Handle = NULL;

            return QItem;
          }
      }
    return NULL;
  }

/* ------------------------------------------------------------------- */

static void QueuePoolRelease (Que, QItem)
    Queue * Que;
    QueueItem * QItem;
  {
    if (QItem->Handle != NULL)
      {
        _HANDLE_DESTROY (QItem->Handle);
      }

    QItem->Allocated = FALSE;
  }

/* ------------------------------------------------------------------- */

static Queue * QueueCreate (Length)
    QueueIndex Length;
  {
    QueueIndex QIndex;
    Queue * Que;

    Que = (Queue *) _MALLOC (sizeof (Queue), "Queue Context");
    Que->Length = Length;
    Que->Head = NULL;
    Que->_Pool = (QueueItem *) _MALLOC (sizeof (QueueItem) * Length,
                "Queue Pool");

    for (QIndex = 0; QIndex < Length; QIndex++)
      {
        QueueItem * QItem = &Que->_Pool[QIndex];
        QItem->Allocated = FALSE;
      }

    return Que;
  }

/* ------------------------------------------------------------------- */

static void QueueDestroy (Que)
    Queue * Que;
  {
    _HANDLE Handle;

    while ((Handle = QueueGetHead (Que)) != NULL)
      {
        _HANDLE_DESTROY (Handle);
      }

    _FREE (Que);
  }

/* -------------------------------------------------------------------- */

static QueueIndex QueueLength (Que)
    Queue * Que;
  {
    return Que->Length;
  }

/* -------------------------------------------------------------------- */

static QueueIndex QueueSize (Que)
    Queue * Que;
  {
    QueueItem * QItem;
    int QSize = 0;

    for (QItem = Que->Head; QItem != NULL; QItem = QItem->Next)
        QSize++;

    return QSize;
  }

/* -------------------------------------------------------------------- */

static _HANDLE QueueGetHead (Que)
    Queue * Que;
  {
    _HANDLE Handle = NULL;

    if (Que->Head != NULL)
      {
        QueueItem * QItem = Que->Head;

        Handle = QItem->Handle;
        QItem->Handle = NULL;
        Que->Head = QItem->Next;

        QueuePoolRelease (Que, QItem);
      }

    return Handle;
  }

/* -------------------------------------------------------------------- */

static _HANDLE QueueGetElement (Que, QOffset)
    Queue * Que;
    QueueIndex QOffset;
  {
    QueueItem * QItem = Que->Head;
    QueueItem * QPrev = NULL;
    QueueIndex QIndex;
    _HANDLE Handle = NULL;

    for (QIndex = 0; QItem != NULL && QIndex < QOffset; QIndex++)
        QPrev = QItem, QItem = QItem->Next;

    if (QItem == NULL)
        return NULL;

    if (QPrev == NULL)
        Que->Head = QItem->Next;
    else
        QPrev->Next = QItem->Next;

    Handle = QItem->Handle;
    QItem->Handle = NULL;
    QueuePoolRelease (Que, QItem);

    return Handle;
  }

/* -------------------------------------------------------------------- */

static _HANDLE QueuePeekElement (Que, QOffset)
    Queue * Que;
    QueueIndex QOffset;
  {
    QueueItem * QItem = Que->Head;
    QueueIndex QIndex;

    for (QIndex = 0; QItem != NULL && QIndex < QOffset; QIndex++)
        QItem = QItem->Next;

    if (QItem == NULL)
        return NULL;

    return QItem->Handle;
  }

/* --------------------------------------------------------------------- */

static Boolean QueueInsert (Que, Handle)
    Queue * Que;
    _HANDLE Handle;
  {
    QueueItem * QItem = QueuePoolCapture (Que);
    if (QItem == NULL)
        return FALSE;

    QItem->Handle = Handle;
    QItem->Next = NULL;

    if (Que->Head == NULL)
      {
        Que->Head = QItem;
      }
    else
      {
        QueueItem * QInsert = Que->Head;

        while (QInsert->Next != NULL)
            QInsert = QInsert->Next;

        QInsert->Next = QItem;
      }

    return TRUE;
  }

/* --------------------------------------------------------------------- */

#ifdef TEST

void main (argc, argv)
    int argc;
    char ** argv;
  {
    Queue * Que;
    char String[100];
    QueueIndex QIndex;

    Que = QueueCreate (100);
    printf ("Length = %u, Size = %u\n", QueueLength (Que), QueueSize (Que));
    for (QIndex = 0; QIndex < 105; QIndex++)
      {
        sprintf (String, "This is a Test! Iteration %d", QIndex);
        if (QueueInsert (Que, strdup (String)) == FALSE)
            printf ("Insert: failed at %u\n", QIndex);
      }
    printf ("Length = %u, Size = %u\n", QueueLength (Que), QueueSize (Que));
    printf ("Head = %s\n", QueueGetHead (Que));
    printf ("Head = %s\n", QueueGetHead (Que));
    printf ("Head = %s\n", QueueGetHead (Que));
    QueueDestroy (Que);
  }

#endif

/* --------------------------------------------------------------------- */
```

2.2.4.8.2. Table

```

/* -------------------------------------------------------------------- */
/* $Id: q_table.c,v 1.1 1995/10/10 07:32:25 mgream Exp $
 * $Log: q_table.c,v $
 * Revision 1.1  1995/10/10  07:32:25  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* These are the different types of options that we can have:
 |  DROPTAIL -- use input policy of drop tail.
 |  DROPRANDOM -- use input policy of random drop.
 |  RED -- use input policy of random early detection.
 |  PRIOSIZE -- use output policy of size priority.
 |  PRIOCLASS -- use output policy of class priority.
 |  ADDRESS -- use output policy of round robin address based fair
 |      queueing.
 */

#   define      QUEUE_OPT_DROPTAIL      (1 << 0)
#   define      QUEUE_OPT_DROPRANDOM    (1 << 1)
#   define      QUEUE_OPT_RED           (1 << 2)
#   define      QUEUE_OPT_PRIOSIZE      (1 << 3)
#   define      QUEUE_OPT_PRIOCLASS     (1 << 4)
#   define      QUEUE_OPT_ADDRESS       (1 << 5)

#   define      QUEUE_OPT_DEFAULT       (QUEUE_OPT_DROPTAIL)

/* -------------------------------------------------------------------- */

/*  QueueEntry
 |
 |  The instance data structure for each queue; contains the queue
 |  itself along with information about options and so on.
 */
typedef struct QueueEntry_ST
  {
    Boolean Allocated;
    int Options;
    /* Space for Policy Context Data */
    /* Space for Extract Context Data */
    int Ext_AddressLast;
    int Ext_SizeLast;
    Queue * Que;
  } QueueEntry;

/* -------------------------------------------------------------------- */

#ifdef DECLARE_MAIN_VARIABLES
#   define      _SCOPE
#else
#   define      _SCOPE                  extern
#endif

#   define      QUEUE_TABLE_SZ          1024

_SCOPE QueueEntry                       QueueTable[QUEUE_TABLE_SZ];
_SCOPE int                              QueueTableInitialised
#ifdef DECLARE_MAIN_VARIABLES
    = 0;
#else
    ;
#endif

/* -------------------------------------------------------------------- */

/*  QueueTableAlloc
 |
 |  Allocate a Queue Entry and place it into the table with all the
 |  appropriate setup details; return the index.
 */
static int QueueTableAlloc (QueueIndex Length, int Options)
  {
    int QIndex;

    if (QueueTableInitialised == 0)
      {
        QueueTableInitialised = 1;
        for (QIndex = 0; QIndex < QUEUE_TABLE_SZ; QIndex++)
```

```
            QueueTable[QIndex].Allocated = FALSE;
      }

    for (QIndex = 0; QueueTable[QIndex].Allocated == TRUE &&
         QIndex < QUEUE_TABLE_SZ; QIndex++)
        ;

    if (QIndex < QUEUE_TABLE_SZ)
      {
        QueueEntry * QEntry = &QueueTable[QIndex];

        QEntry->Allocated = TRUE;
        QEntry->Options = Options;
        QEntry->Ext_AddressLast = 0;
        QEntry->Ext_SizeLast = 0;
        QEntry->Que = QueueCreate (Length);
      }

    return QIndex;
  }

/* -------------------------------------------------------------------- */

/*  QueueTableFree
 |
 |  Free up the specified entry.
 */
static void QueueTableFree (int QIndex)
  {
    QueueEntry * QEntry = &QueueTable[QIndex];

    if (QEntry->Allocated == TRUE)
      {
        QEntry->Allocated = FALSE;

        QueueDestroy (QEntry->Que);
      }
  }

/* -------------------------------------------------------------------- */

```

2.3. Transport Layer

2.3.1. Data Structures

2.3.1.1. IE Transport Primitive

This Data Structure has no content.

2.3.1.2. IE Transport Parameters

Name	*Type*	*Subrange*	*Default Value*
Initial Sequence Number	INTEGER	(-Inf,+Inf)	37089

2.3.1.3. Msg Transport Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.3.1.4. Msg Transport Connect Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Address	INTEGER	[0,512)	0

2.3.1.5. Msg Transport Connect Request

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Address	*INTEGER*	*[0,512)*	*0*

2.3.1.6. Msg Transport Data Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	Msg Primitive		

2.3.1.7. Msg Transport Data Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	*Msg Primitive*		

2.3.1.8. Msg Transport Data Request

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Content	*Msg Primitive*		

2.3.1.9. Msg Transport Disconnect Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.3.1.10. Msg Transport Disconnect Request

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

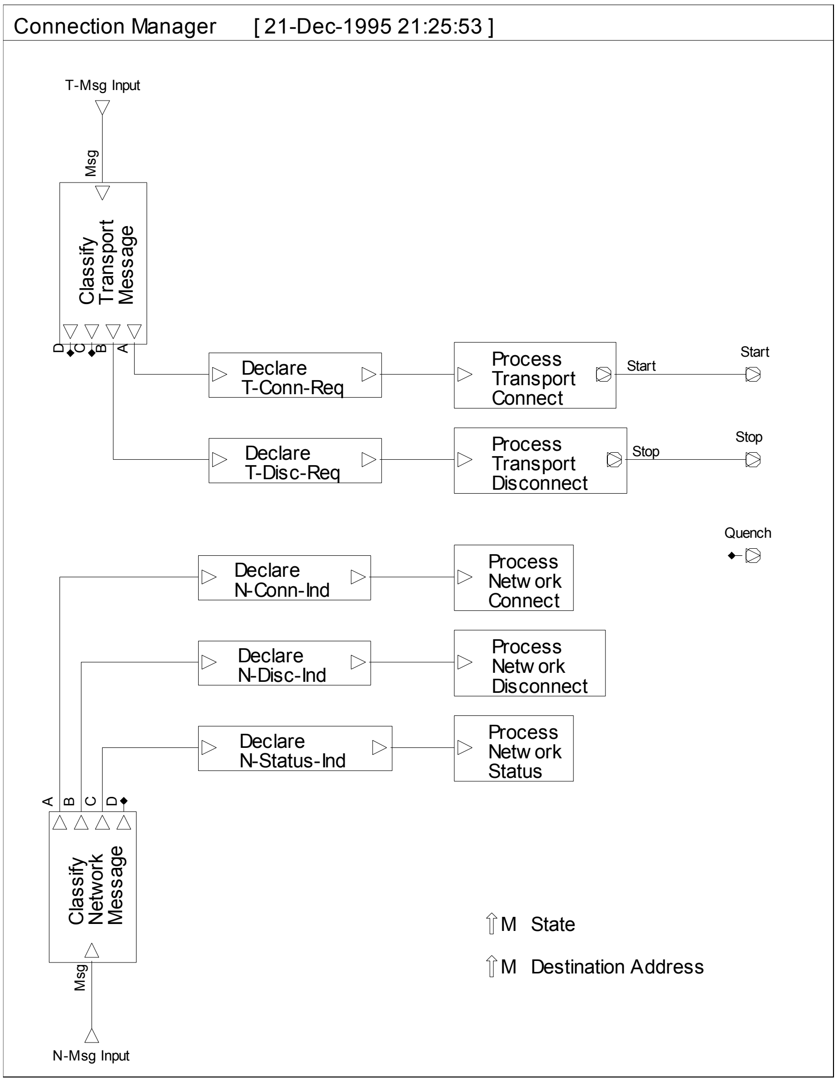
2.3.1.11. Msg Transport TCP

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Seq	*INTEGER*	*(-Inf,+Inf)*	*0*
Ack	*INTEGER*	*(-Inf,+Inf)*	*0*
Win	*INTEGER*	*(-Inf,+Inf)*	*0*
Flag--Ack	*Boolean*		*False*
T_Now	*INTEGER*	*(-Inf,+Inf)*	*0*
T_Recent	*INTEGER*	*(-Inf,+Inf)*	*0*
Flag--Timestamp	*Boolean*		*False*

2.3.2. Main Modules

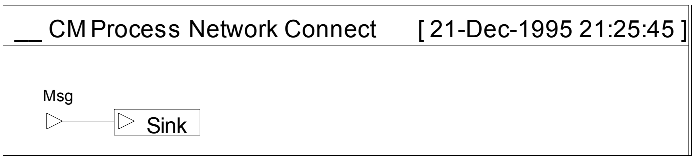
2.3.2.1. Connection Manager

This Module implements “DFD 1: Connection Manager”.



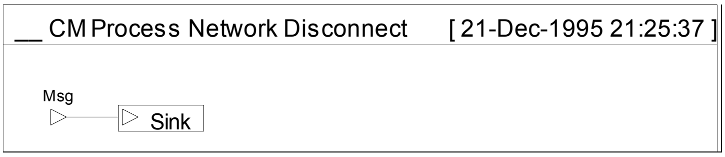
2.3.2.2. Connection Manager -- Process Network Connect

This Module implements “PSPEC 1.2: Process Network Connect”.



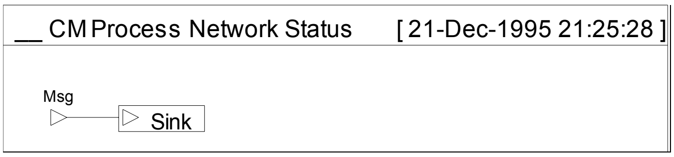
2.3.2.3. Connection Manager -- Process Network Disconnect

This Module implements “PSPEC 1.3: Process Network Disconnect”.



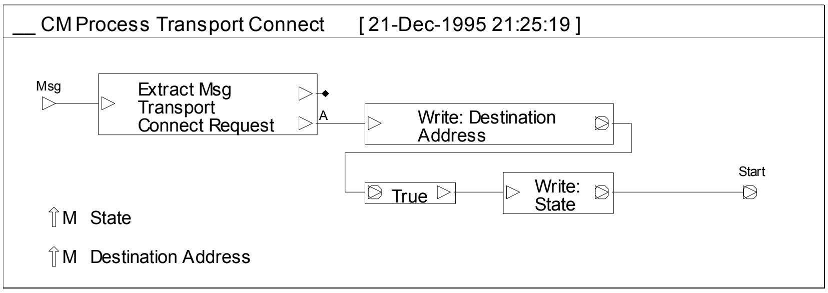
2.3.2.4. Connection Manager -- Process Network Status

This Module implements “PSPEC 1.4: Process Network Status”.



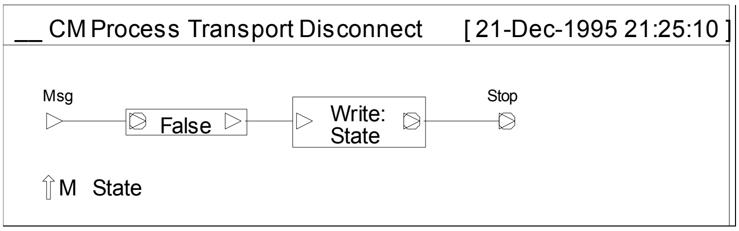
2.3.2.5. Connection Manager -- Process Transport Connect

This Module implements “PSPEC 1.6: Process Transport Connect”.


__ CM Process Transport Connect [21-Dec-1995 21:25:19]
Msg
Extract Msg Transport Connect Request
A
Write: Destination Address
True
Write: State
Start
M State
M Destination Address

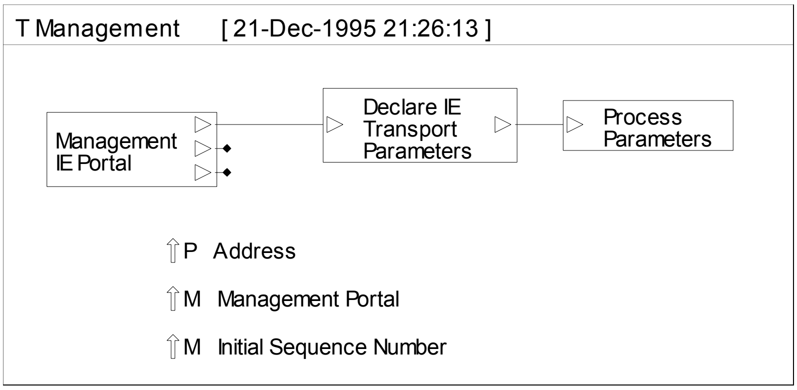
2.3.2.6. Connection Manager -- Process Transport Disconnect

This Module implements “PSPEC 1.7: Process Transport Disconnect”.



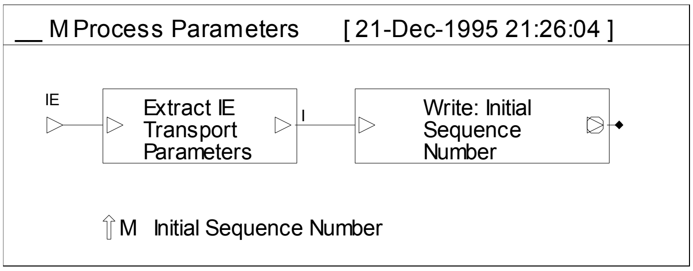
2.3.2.7. Management

This Module implements “DFD 2: Management Processor”. This also incorporates “PSPEC 2.1: Validate Mgmt Message and Extract IE”.


T Management [21-Dec-1995 21:26:13]
Management IE Portal
Declare IE Transport Parameters
Process Parameters
P Address
M Management Portal
M Initial Sequence Number

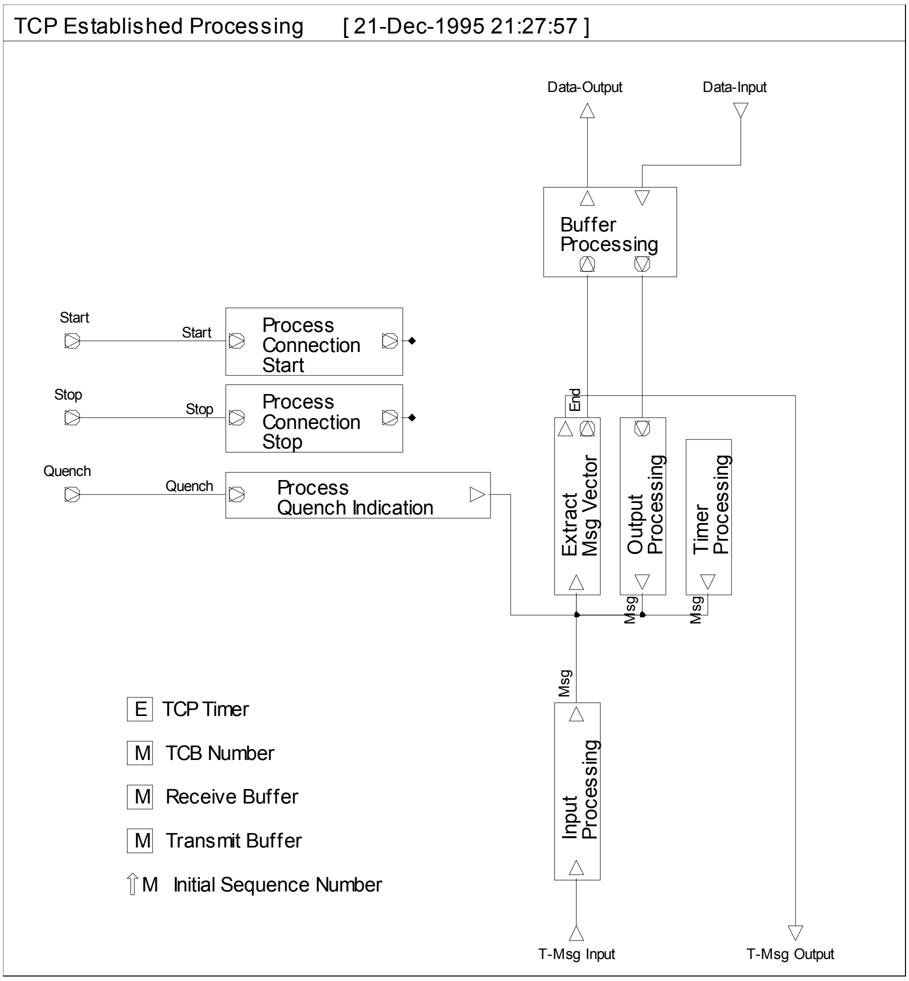
2.3.2.8. Management -- Process Parameters

This Module implements “PSPEC 2.2: Process Setup IE”.


__ M Process Parameters [21-Dec-1995 21:26:04]
IE
Extract IE Transport Parameters
I
Write: Initial Sequence Number
M Initial Sequence Number

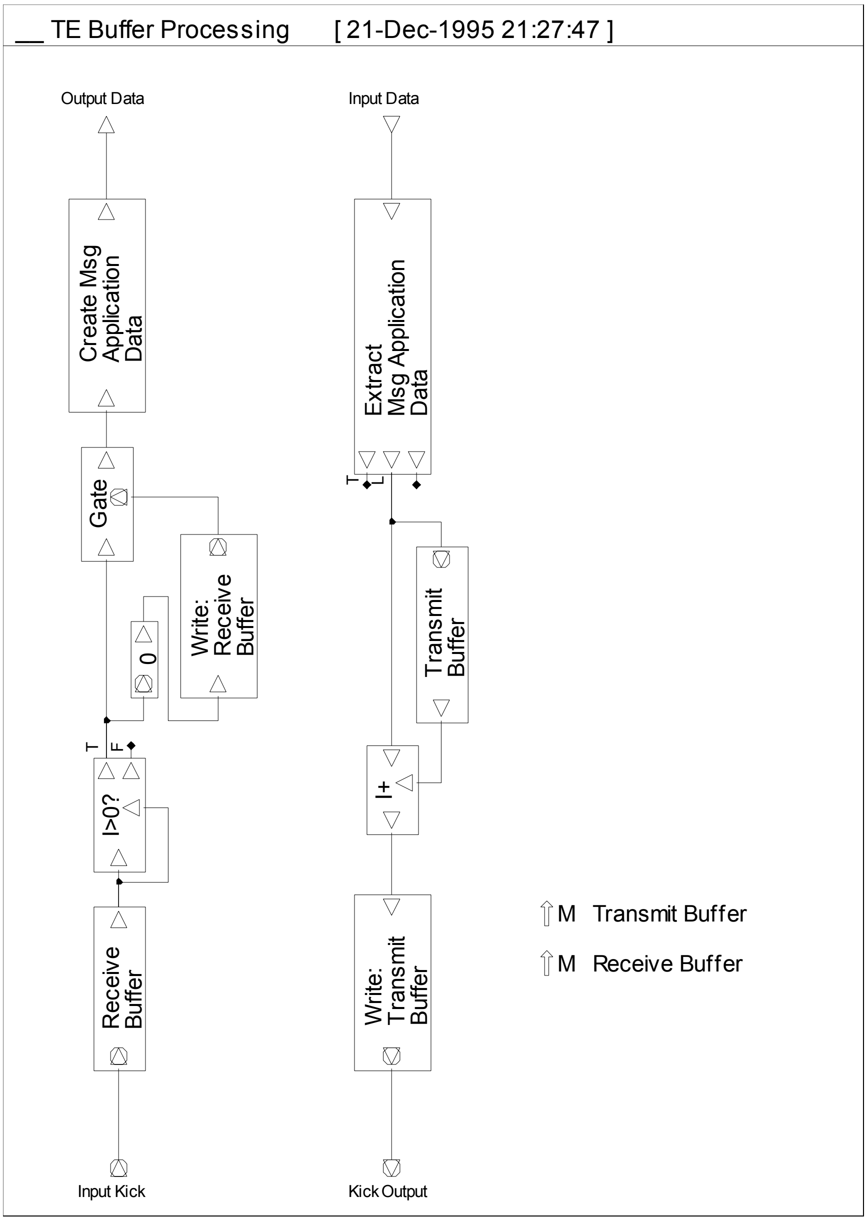
2.3.2.9. TCP Established Processing

This Module implements “DFD 3: TCP Processing”. In this diagram, the module names have been renamed such that “Process Connection Start” corresponds to “TCP Start”; “Process Connection Stop” to “TCP Stop”; and “Process Quench Indication” to “TCP Quench”.



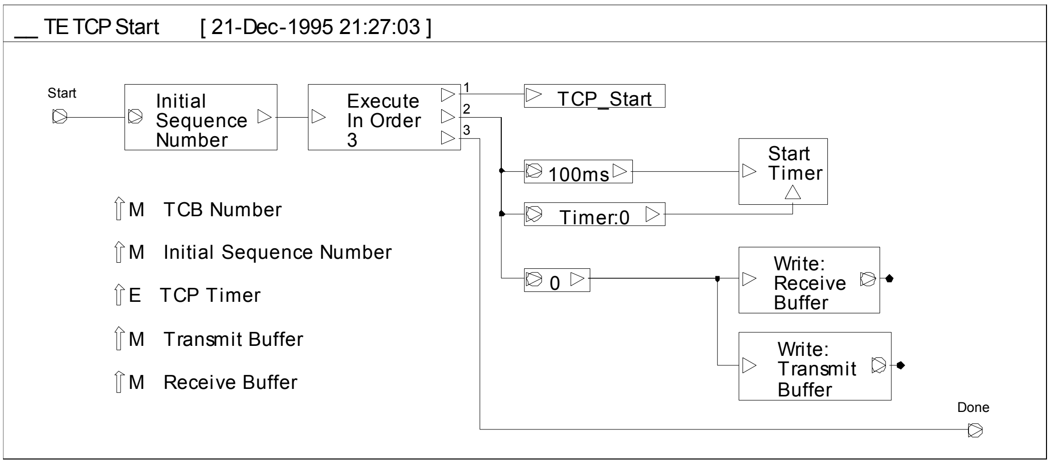
2.3.2.10. TCP Established Processing -- Buffer Processing

This Module was an implict part of "PSPEC 5.4: Process TCP Outgoing" but has been factored into a separate module.



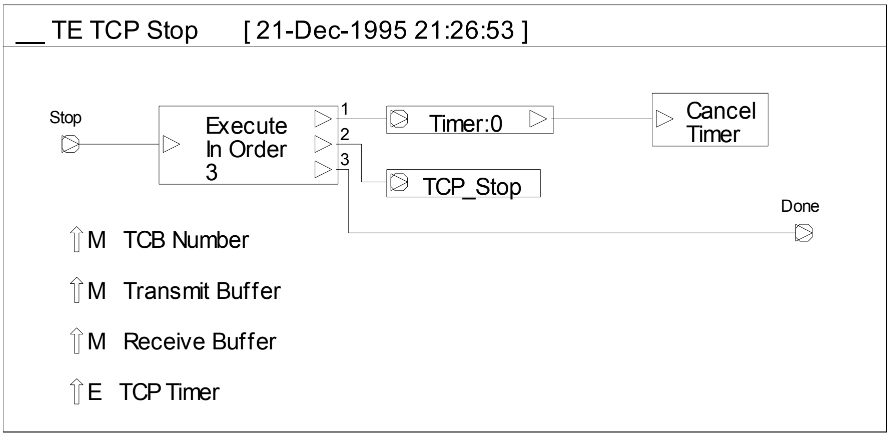
2.3.2.11. TCP Established Processing -- TCP Start

This Module implements “PSPEC 3.1: Start TCP”.



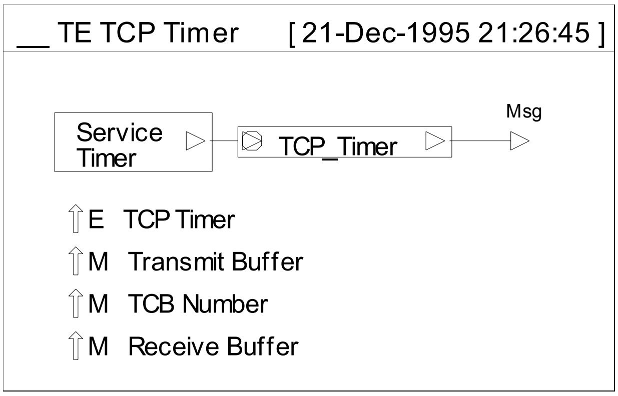
2.3.2.12. TCP Established Processing -- TCP Stop

This Module implements “PSPEC 3.2: Stop TCP”.



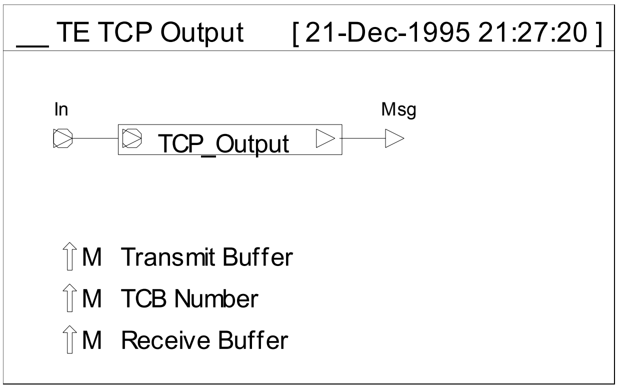
2.3.2.13. TCP Established Processing -- TCP Timer

This Module implements “PSPEC 3.3: Process TCP Timers”.



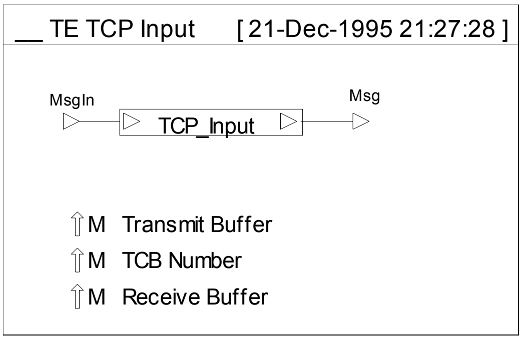
2.3.2.14. TCP Established Processing -- TCP Output

This Module implements “PSPEC 3.4: Process TCP Outgoing”.



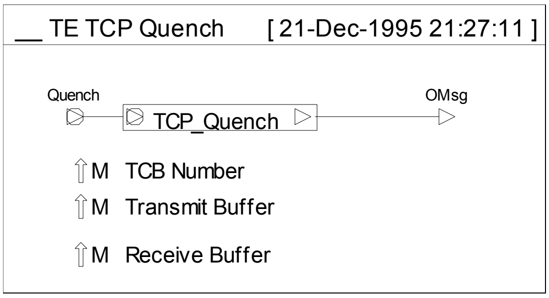
2.3.2.15. TCP Established Processing -- TCP Input

This Module implements “PSPEC 3.5: Process TCP Incoming”.



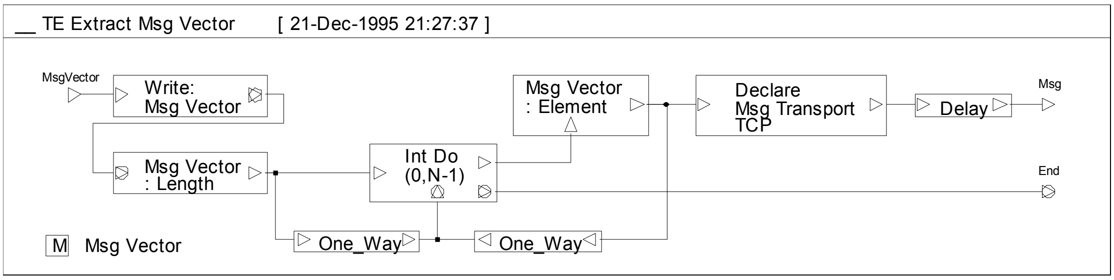
2.3.2.16. TCP Established Processing -- TCP Quench

This Module implements an element not present in the design; it was later determined that the TCP Queue facility should be added for potential investigation.



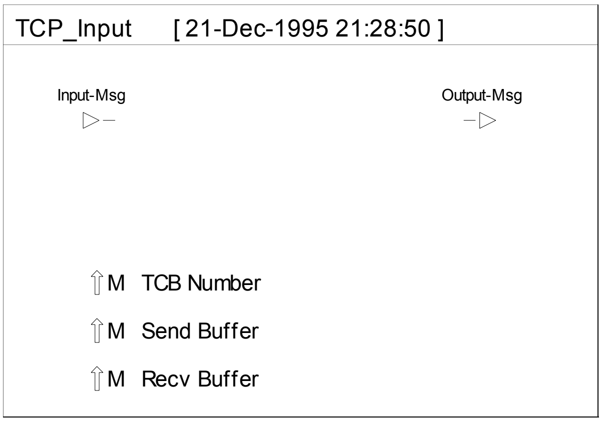
2.3.2.17. TCP Established Processing -- Extract Msg Vector

This Module implements "PSPEC 3.6: Transmit TCP Messages". Due to the nature of the 'C' implementation, a mechanism for allowing the release of multiple Messages needed to be constructed; this module extracts those Messages and sends them.



2.3.2.18. TCP Input

This Module acts as the interface betwen BONeS and the *TCP Module*, as implemented in 'C'for processing a received TCP segment. *The TCB Number* correponds to an entry in a TCB Table allocated from *TCP Start,* the *Send* and *Recv Buffers* are self descriptive.

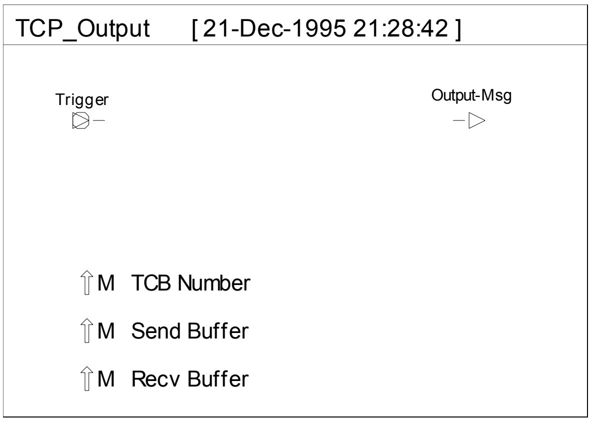


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* --------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_TCP_Input.c"
/* --------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User RUN Below Here */

/* --------------------------------------------------------------------- */
    BONeS_TCP_Input (Input_Msg, Output_Msg, argvector);
/* --------------------------------------------------------------------- */

    /* User RUN Above Here */

```

2.3.2.19. TCP Output

This Module acts as the interface betwen BONeS and the *TCP Module*, as implemented in ‘C’for processing an outgoing unit of data. *The TCB Number correponds to an entry in a TCB Table allocated from TCP Start, the Send and Recv Buffers are self descriptive.*

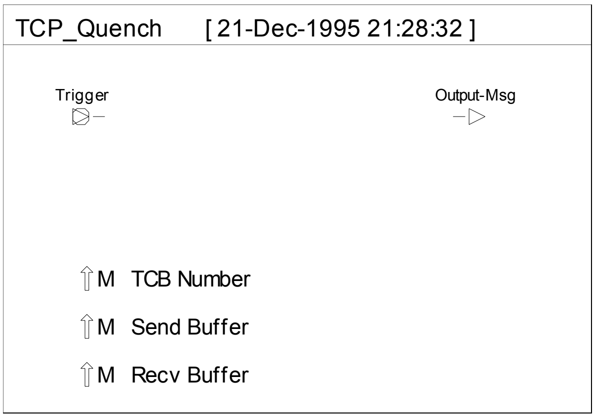


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_TCP_Output.c"
/* -------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User RUN Below Here */

/* -------------------------------------------------------------------- */
    BONeS_TCP_Output (Trigger, Output_Msg, argvector);
/* -------------------------------------------------------------------- */

    /* User RUN Above Here */

```

2.3.2.20. TCP Quench

This Module acts as the interface betwen BONeS and the *TCP Module*, as implemented in 'C'for processing a network indication of traffic congestion. *The TCB Number correponds to an entry in a TCB Table allocated from TCP Start, the Send and Recv Buffers are self descriptive.*

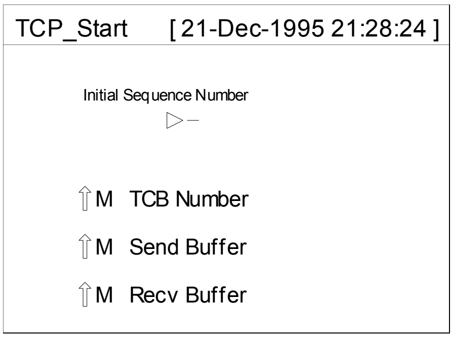


Extracts of the 'C' interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_TCP_Quench.c"
/* -------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User RUN Below Here */

/* -------------------------------------------------------------------- */
    BONeS_TCP_Quench (Trigger, Output_Msg, argvector);
/* -------------------------------------------------------------------- */

    /* User RUN Above Here */

```

2.3.2.21. TCP Start

This Module acts as the interface betwen BONeS and the *TCP Module*, as implemented in 'C'for starting TCP processing. *The TCB Number correponds to an* *entry in a TCB Table allocated by this module, the Send and Recv Buffers are self descriptive.*

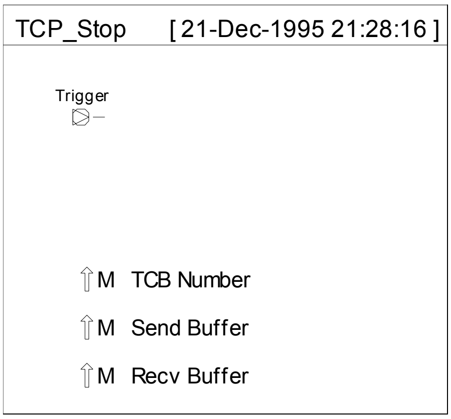


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_TCP_Start.c"
/* -------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User RUN Below Here */

/* -------------------------------------------------------------------- */
    BONeS_TCP_Start (InitialSequenceNumber, argvector);
/* -------------------------------------------------------------------- */

    /* User RUN Above Here */

```

2.3.2.22. TCP Stop

This Module acts as the interface betwen BONeS and the *TCP Module*, as implemented in ‘C’for stopping TCP processing. *The TCB Number correponds to an entry in a TCB Table allocated from TCP Start, the Send and Recv Buffers are self descriptive.*

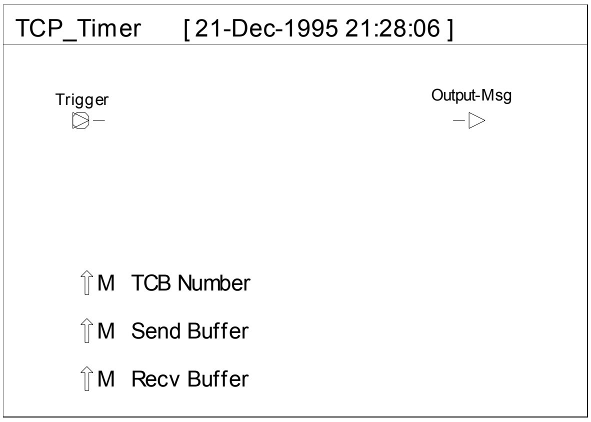


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_TCP_Stop.c"
/* -------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User RUN Below Here */

/* -------------------------------------------------------------------- */
    BONeS_TCP_Stop (Trigger, argvector);
/* -------------------------------------------------------------------- */

    /* User RUN Above Here */

```

2.3.2.23. TCP Timer

This Module acts as the interface betwen BONeS and the *TCP Module*, as implemented in ‘C’for processing the periodic TCP Timer. *The TCB Number correponds to an entry in a TCB Table allocated from TCP Start, the Send and Recv Buffers are self descriptive.*

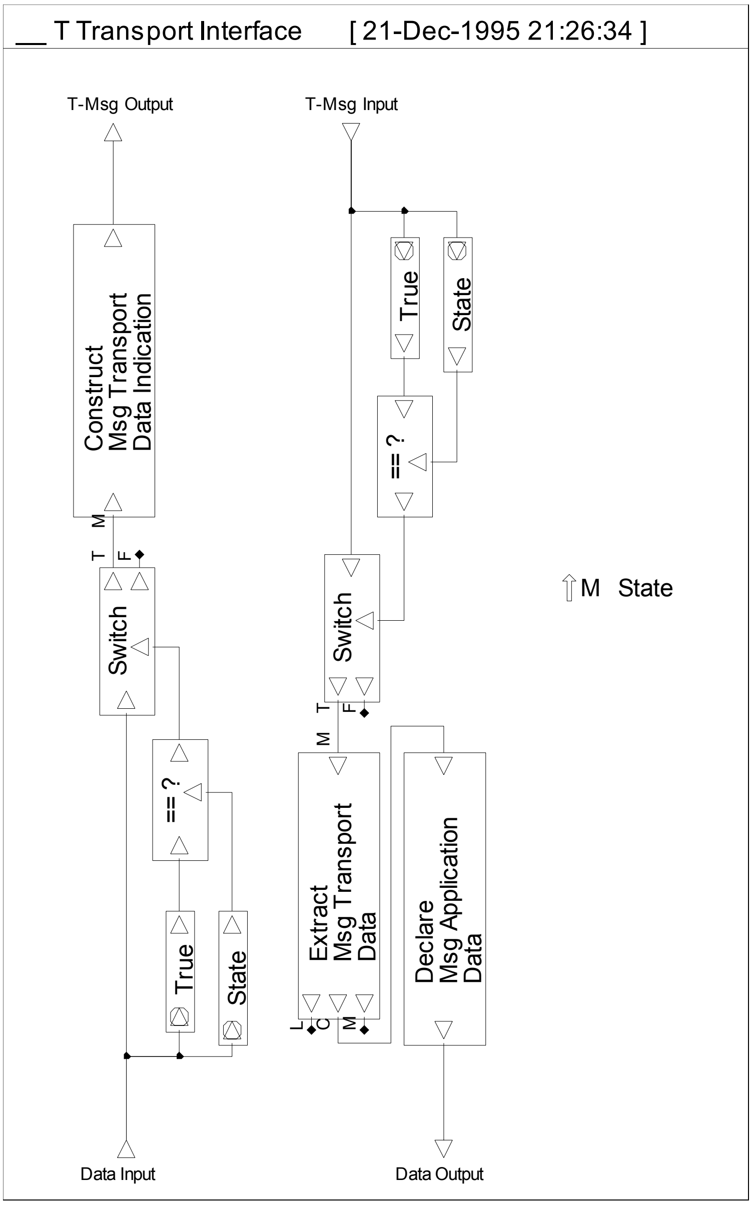


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_TCP_Timer.c"
/* -------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User RUN Below Here */

/* -------------------------------------------------------------------- */
    BONeS_TCP_Timer (Trigger, Output_Msg, argvector);
/* -------------------------------------------------------------------- */

    /* User RUN Above Here */

```

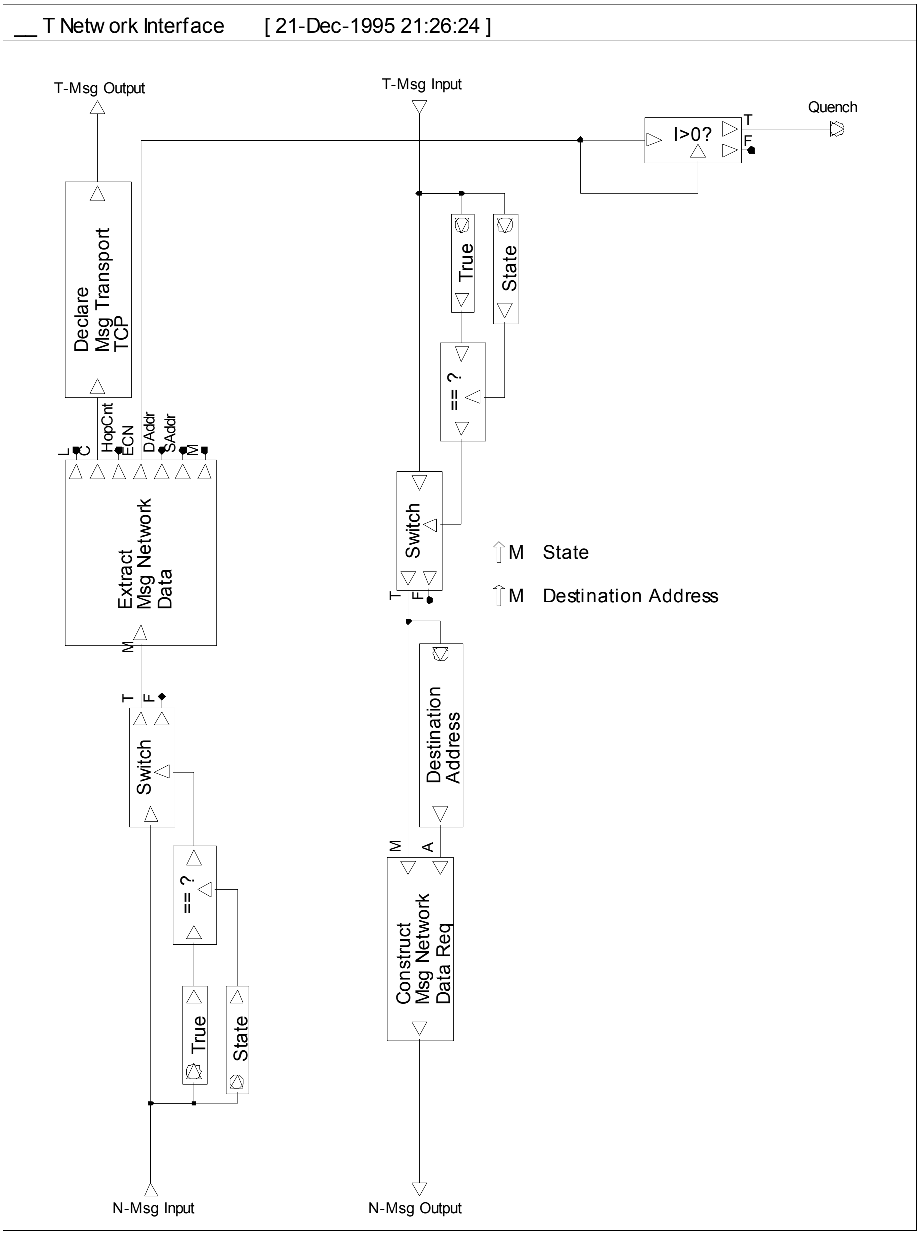
2.3.2.24. Transport Interface

This Module implements “DFD 4: Transport Interface”. This also includes “PSPEC 4.1: Process Outgoing Data” and “PSPEC 4.2: Process Incoming Data”.



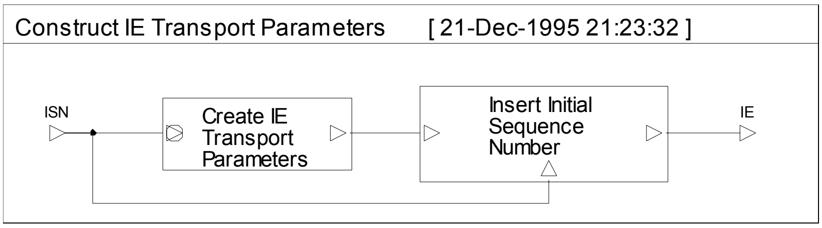
2.3.2.25. Network Interface

This Module implements "DFD 5: Network Interface". This also includes "PSPEC 5.1: Process Incoming Message" and "PSPEC 5.2: Process Outgoing Message".

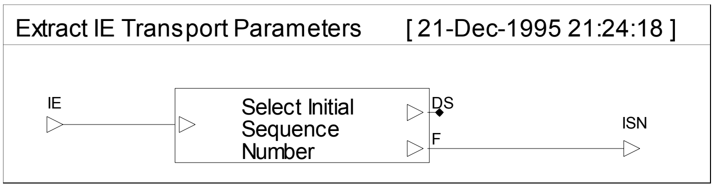


2.3.3. Support Modules

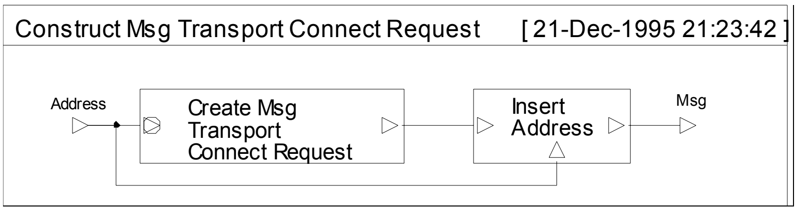
2.3.3.1. Construct IE Transport Parameters


Construct IE Transport Parameters [21-Dec-1995 21:23:32]
ISN
Create IE Transport Parameters
Insert Initial Sequence Number
IE

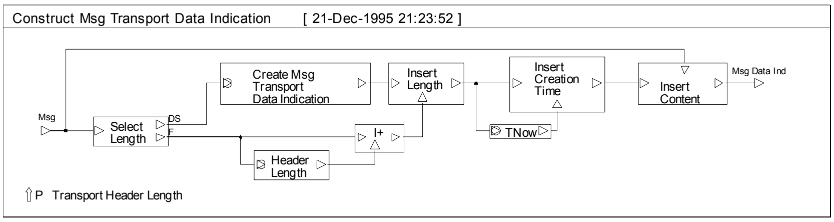
2.3.3.2. Extract IE Trasnport Parameters


Extract IE Transport Parameters [21-Dec-1995 21:24:18]
IE
Select Initial Sequence Number
DS
F
ISN

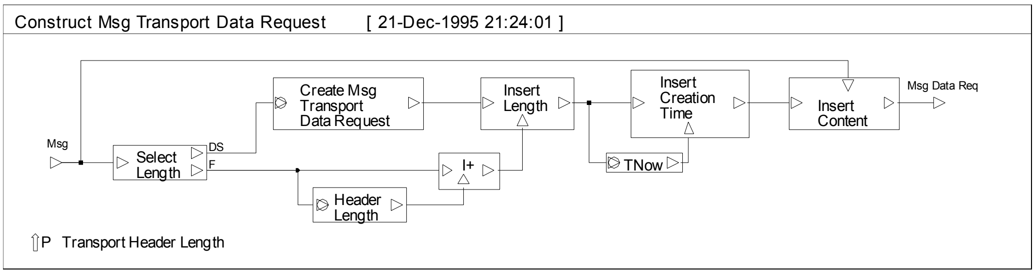
2.3.3.3. Construct Msg Transport Connect Request


Construct Msg Transport Connect Request [21-Dec-1995 21:23:42]
Address
Create Msg Transport Connect Request
Insert Address
Msg

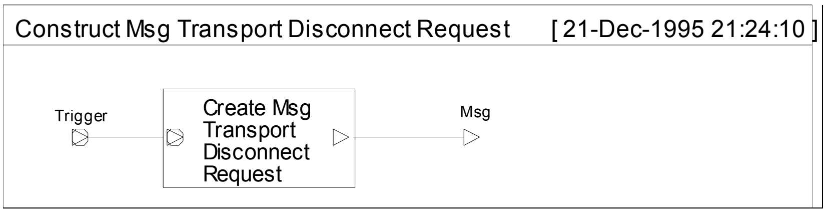
2.3.3.4. Construct Msg Transport Data Indication


Construct Msg Transport Data Indication [21-Dec-1995 21:23:52]
Msg
Select Length
DS
F
Create Msg Transport Data Indication
Insert Length
I+
Header Length
Insert Creation Time
TNow
Insert Content
Msg Data Ind
P Transport Header Length

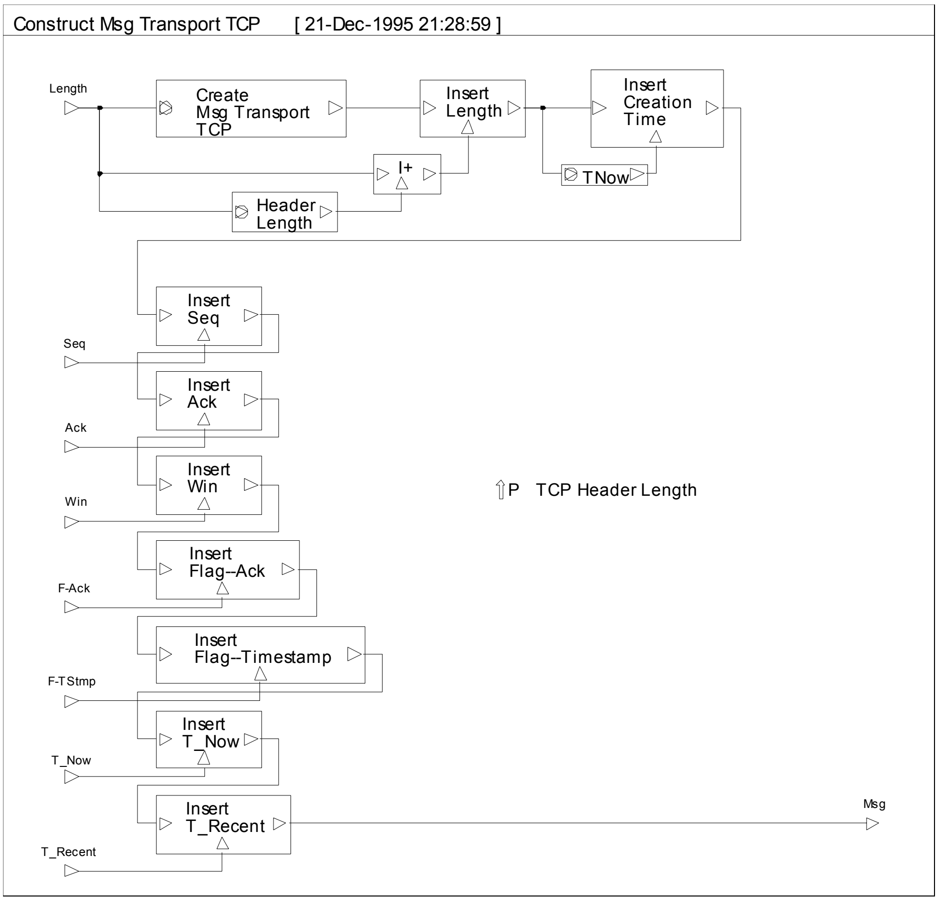
2.3.3.5. Construct Msg Transport Data Request


Construct Msg Transport Data Request [21-Dec-1995 21:24:01]
Msg
Select Length
DS
F
Create Msg Transport Data Request
Insert Length
I+
Header Length
Insert Creation Time
TNow
Insert Content
Msg Data Req
P Transport Header Length

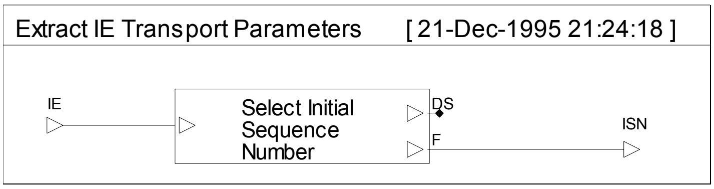
2.3.3.6. Construct Msg Transport Disconnect Request


Construct Msg Transport Disconnect Request [21-Dec-1995 21:24:10]
Trigger
Create Msg Transport Disconnect Request
Msg

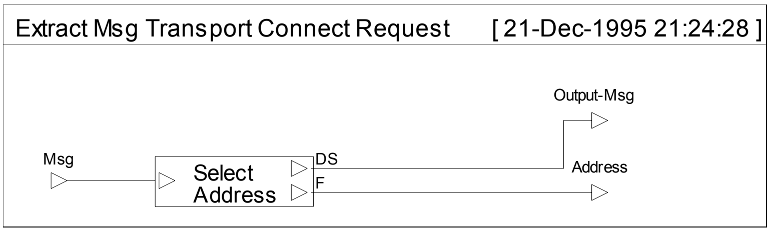
2.3.3.7. Construct Msg Transport TCP



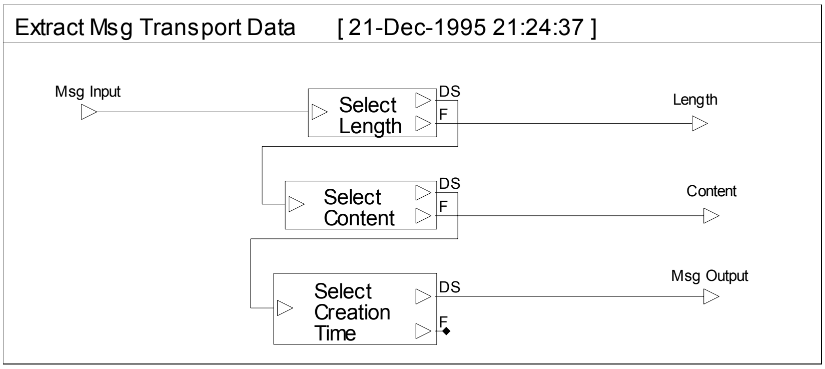
2.3.3.8. Extract IE Transport Parameters


Extract IE Transport Parameters [21-Dec-1995 21:24:18]
IE
Select Initial Sequence Number
DS
F
ISN

2.3.3.9. Extract Msg Transport Connect Request


Extract Msg Transport Connect Request [21-Dec-1995 21:24:28]
Output-Msg
Msg
Select Address
DS
F
Address

2.3.3.10. Extract Msg Transport Data



2.3.4. ‘C’ Modules

The Transmission Control Protocol (TCP) implementation was carried out by implementing bridging functions that mapped between the TCP implementation and BONeS. This decoupling was for the purposes of allowing testing of the TCP functionality outside of the BONeS environment.

2.3.4.1. BONeS Interface

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_Interface.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 * $Log: BONeS_Interface.c,v $
 * Revision 1.2  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/*  _BONeS_Get_Send_Buffer_Sz
 |
 |  Return the size of the current send buffer; that is supplied as an
 |  argument to this module.
 */
static int _BONeS_Get_Send_Buffer_Sz (Tcb)
    TcbPtr Tcb;
  {
    arg_ptr argvector = Tcb->argvector;
    return __GetINTEGERVal (SendBuffer_arc);
  }

/* -------------------------------------------------------------------- */

/*  _BONeS_Set_Send_Buffer_Sz
 |
 |  Set the size of the send buffer according to the passed argument.
 */
static int _BONeS_Set_Send_Buffer_Sz (Tcb, Size)
    TcbPtr Tcb;
    int Size;
  {
    arg_ptr argvector = Tcb->argvector;
    __PutINTEGERVal (SendBuffer_arc, Size);
    return __GetINTEGERVal (SendBuffer_arc);
  }

/* -------------------------------------------------------------------- */

/*  _BONeS_Get_Recv_Buffer_Sz
 |
 |  Return the size of the receive buffer, that is supplied as a BONeS
 |  argument to this primitive.
 */
static int _BONeS_Get_Recv_Buffer_Sz (Tcb)
    TcbPtr Tcb;
  {
    arg_ptr argvector = Tcb->argvector;
    return __GetINTEGERVal (RecvBuffer_arc);
  }

/* ------------------------------------------------------------------- */

/*  _BONeS_Set_Recv_Buffer_Sz
 |
 |  Set the size of the recv buffer.
 */
static int _BONeS_Set_Recv_Buffer_Sz (Tcb, Size)
    TcbPtr Tcb;
    int Size;
  {
    arg_ptr argvector = Tcb->argvector;
    __PutINTEGERVal (RecvBuffer_arc, Size);
    return __GetINTEGERVal (RecvBuffer_arc);
  }

/* ------------------------------------------------------------------- */

```

2.3.4.2. BONeS TCP Start

```

/* ------------------------------------------------------------------- */
/* $Id: BONeS_TCP_Start.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 * $Log: BONeS_TCP_Start.c,v $
 * Revision 1.2  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* ------------------------------------------------------------------- */

/* ------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/TCP.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_Interface.c"
/* ------------------------------------------------------------------- */

/*  BONeS_TCP_Start
 |
 |  Setup a TCB; by creating one and setting up the index mapping to
 |  it, and then initialising the sequence numbers with the passed
 |  initial sequence number.
 */
static void BONeS_TCP_Start (InitialSequenceNumber, argvector)
    arc_ptr InitialSequenceNumber;
    arg_ptr argvector;
  {
    int _TcbNumber = TcbCreate ();
    TcbPtr Tcb = TcbLookup (TCBNumber);

    __PutINTEGERVal (TCBNumber_arc, _TcbNumber);

    if (Tcb != NULL)
      {
        Init_Process (Tcb, __GetINTEGERVal (InitialSequenceNumber));
      }

    __FreeArc (InitialSequenceNumber);
  }

/* ------------------------------------------------------------------- */

```

2.3.4.3. BONeS TCP Stop

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_TCP_Stop.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 * $Log: BONeS_TCP_Stop.c,v $
 * Revision 1.2  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/TCP.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_Interface.c"
/* -------------------------------------------------------------------- */

/*  BONeS_TCP_Stop
 |
 |  Time to shut down the TCB; which is done simply by calling the
 |  tcb destroy mechanism. There is nothing else to clean up, but
 |  we do make sure to set the tcbnumber index to an invalid value
 |  so that it can't be accidently reused.
 */
static void BONeS_TCP_Stop (InTrigger, argvector)
    arc_ptr InTrigger;
    arg_ptr argvector;
  {
    TcbDestroy (TCBNumber);

    __PutINTEGERVal (TCBNumber_arc, -1);
    __FreeArc (InTrigger);
  }

/* -------------------------------------------------------------------- */

```

2.3.4.4. BONeS TCP Input

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_TCP_Input.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 * $Log: BONeS_TCP_Input.c,v $
 * Revision 1.2  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/TCP.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_Interface.c"
/* -------------------------------------------------------------------- */

/*  BONeS_TCP_Input
 |
 |  The input processing is slightly tricky because we have an input
 |  message that must be converted into an internal representation,
 |  which we do first before firing up the input process. Again, as
 |  with other TCP processing, we have the required output queue
 |  setup and extraction.
 */
static void BONeS_TCP_Input (InputMsg, OutputMsg, argvector)
    arc_ptr InputMsg;
    arc_ptr OutputMsg;
    arg_ptr argvector;
  {
    TcbPtr Tcb = TcbLookup (TCBNumber);
    MsgPtr Msg;

    if (Tcb != NULL)
      {
        Tcb->argvector = argvector;
        OutQueue_Initialise ();
        Msg = MsgCreate ();
        MsgConvertFromBONeS (Msg, InputMsg);
        Input_Process (Tcb, Msg);
        MsgDestroy (Msg);
        OutQueue_Extract (OutputMsg);
      }

    __FreeArc (InputMsg);
  }

/* -------------------------------------------------------------------- */

```

2.3.4.5. BONeS TCP Output

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_TCP_Output.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 * $Log: BONeS_TCP_Output.c,v $
 * Revision 1.2  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/TCP.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_Interface.c"
/* -------------------------------------------------------------------- */

/*  BONeS_TCP_Output
 |
 |  We come into here on a kick as well; the output process fires
 |  in a non-forced mode after having set up the output queue, upon
 |  exit, the queue is extracted into the output vector and pumped
 |  outwards.
 */
static void BONeS_TCP_Output (InTrigger, OutputMsg, argvector)
    arc_ptr InTrigger;
    arc_ptr OutputMsg;
    arg_ptr argvector;
  {
    TcbPtr Tcb = TcbLookup (TCBNumber);

    if (Tcb != NULL)
      {
        Tcb->argvector = argvector;
        OutQueue_Initialise ();
        Output_Process (Tcb, FALSE);
        OutQueue_Extract (OutputMsg);
      }

    __FreeArc (InTrigger);
  }

/* -------------------------------------------------------------------- */

```

2.3.4.6. BONeS TCP Quench

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_TCP_Quench.c,v 1.1 1995/12/21 11:08:30 mgream Exp $
 * $Log: BONeS_TCP_Quench.c,v $
 * Revision 1.1  1995/12/21  11:08:30  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/TCP.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_Interface.c"
/* -------------------------------------------------------------------- */

/*  BONeS_TCP_Quench
 |
 |  We come into here on a kick as well; the queue process simply
 |  plays with variables and doesn't do much else.
 */
static void BONeS_TCP_Quench (InTrigger, OutputMsg, argvector)
    arc_ptr InTrigger;
    arc_ptr OutputMsg;
    arg_ptr argvector;
  {
    TcbPtr Tcb = TcbLookup (TCBNumber);

    if (Tcb != NULL)
      {
        Tcb->argvector = argvector;
        OutQueue_Initialise ();
        Quench_Process (Tcb);
        OutQueue_Extract (OutputMsg);
      }

    __FreeArc (InTrigger);
  }

/* -------------------------------------------------------------------- */

```

2.3.4.7. BONeS TCP Timer

```

/* -------------------------------------------------------------------- */
/* $Id: BONeS_TCP_Timer.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 * $Log: BONeS_TCP_Timer.c,v $
 * Revision 1.2  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/TCP/TCP.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/BONeS_Interface.c"
/* -------------------------------------------------------------------- */

/*  BONeS_TCP_Timer
 |
 |  The entry point here is a trigger that pumps out every 100ms, we
 |  locate the appropriate Tcb, setup the output queue and then go
 |  and kick the timer process. Upon return, the output queue is
 |  extracted for any messages that have to be pumped outwards. We
 |  also must free the input trigger.
 */
static void BONeS_TCP_Timer (InTrigger, OutputMsg, argvector)
    arc_ptr InTrigger;
    arc_ptr OutputMsg;
    arg_ptr argvector;
  {
    TcbPtr Tcb = TcbLookup (TCBNumber);

    if (Tcb != NULL)
      {
        Tcb->argvector = argvector;
        OutQueue_Initialise ();
        Timer_Process (Tcb);
        OutQueue_Extract (OutputMsg);
      }

    __FreeArc (InTrigger);

  }

/* ------------------------------------------------------------------- */

```

2.3.4.8. BONeS TCP (Primitive)

```

/* ------------------------------------------------------------------- */
/* $Id: TCP.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 * $Log: TCP.c,v $
 * Revision 1.2  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* ------------------------------------------------------------------- */

/* ------------------------------------------------------------------- */

#   include     "/u/mgream/BONeS/Constructed/TCP/prototypes.h"

#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_frag.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_data.c"

#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_tcb.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_msg.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_outqueue.c"

#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_timers.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_output.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_input.c"
#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_quench.c"

#   include     "/u/mgream/BONeS/Constructed/TCP/tcp_init.c"

/* ------------------------------------------------------------------- */

```

2.3.4.8.1. Data

```

/* ------------------------------------------------------------------- */
/* $Id: tcp_data.c,v 1.3 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_data.c,v $
 * Revision 1.3  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.2  1995/10/10  08:15:17  mgream
 * cosmetic changes
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* ------------------------------------------------------------------- */

/* ------------------------------------------------------------------- */
/*  - - - DATA STRUCTURES - - -
 |
 |  There are two primary datastructures in use; the first is the
 |  Transmission Control Block which maintains TCP Information. It
 |  contains information such as session state, fragment queues and
 |  so on. The second data structure is the message, which has a fixed
 |  set of fields and a next pointer so that we can chain them in a
 |  linked list.
 |  There are also macros and defines to cover specific TCP magic
 |  numbers and so forth.
 */
/* -------------------------------------------------------------------- */

typedef int boolean;

#   define      FALSE           0
#   define      TRUE            1

/* -------------------------------------------------------------------- */

typedef unsigned long tcp_seq;

/* -------------------------------------------------------------------- */

/*  TcbEntry
 |
 |  The transmission control block contains the necessary state
 |  information for a TCP instance; this is modelled on the BSD
 |  4.4/Net 3 paradigm.
 */

typedef struct TcbEntry_ST
  {
    /* Flags */
    boolean Flag_DelayedAck;
    boolean Flag_Ack;

    /* Timer variables */
    u_long Timer_Persist;
    u_long Timer_Retransmit;

    /* Send Window/Sequence state */
    u_long snd_wnd;
    tcp_seq snd_una;
    tcp_seq snd_nxt;
    tcp_seq snd_wl1;
    tcp_seq snd_wl2;

    tcp_seq snd_max;

    /* Recv Window/Sequence state */
    u_long rcv_wnd;
    tcp_seq rcv_nxt;

    tcp_seq rcv_adv;

    /* Congestion Control parameters */
    u_long snd_cwnd;
    u_long snd_ssthresh;

    /* Round Trip Time parameters */
    short t_idle;
    short t_rtt;
    tcp_seq t_rtseq;
    short t_srtt;
    short t_rttvar;
    u_short t_rttmin;
    u_long max_sndwnd;

    /* Window Scaling */
    u_char snd_scale;
    u_char rcv_scale;

    /* Timestamp */
    boolean Flag_Timestamp;
    u_long ts_recent;
    u_long ts_recent_age;

    /* Current TCP Time */
    u_long tcp_now;

    /* Retransmit parameters */
    short t_rxtshift;
    short t_rxtcur;
    short t_dupacks;
    u_short t_maxseg;

    tcp_seq last_ack_sent;

    /* Our Stuff */
    int _timer_ticks;
    boolean Allocated;
    Queue * FragQueue;

    arg_ptr argvector;

  } TcbEntry;
typedef TcbEntry * TcbPtr;

/* -------------------------------------------------------------------- */

/*  Message
 |
 |  This is an actual TCP message that is used to communicate between
 |  two TCP instances; note that this is an internal representation
 |  that is mapped from the external (BONeS) one.
 */

typedef struct Message_ST
  {
    struct Message_ST * Next;

    /* Length */
    u_short len;

    /* Sequence/Window Information */
    tcp_seq seq;
    tcp_seq ack;
    u_short win;

    /* Flags */
    boolean Flag_Ack;

    /* Timestamp */
    boolean Flag_Timestamp;
    u_long t_recent;
    u_long t_now;

  } Message;
typedef Message * MsgPtr;

/* -------------------------------------------------------------------- */

/*  Macros
 |
 */

#   define  TCP_MSS             512

#   define  TCP_MAXWIN          65535
#   define  TCP_MAX_WINSHIFT    14
#   define  TCP_RTTDFLT         (TCPTV_SRTTDFLT / PR_SLOWHZ)

#   define  TCP_RTT_SHIFT       3
#   define  TCP_RTTVAR_SHIFT    2
#   define  TCP_RTTVAR_SCALE    4
#   define  TCP_MAXRXTSHIFT     12

#   define  PR_SLOWHZ           2

#   define  TCPTV_MIN           (1  * PR_SLOWHZ)
#   define  TCPTV_REXMTMAX      (64 * PR_SLOWHZ)
#   define  TCPTV_SRTTBASE      (0  * PR_SLOWHZ)
#   define  TCPTV_SRTTDFLT      (3  * PR_SLOWHZ)

#   define  TCPTV_PERSMIN       (5  * PR_SLOWHZ)
#   define  TCPTV_PERSMAX       (60 * PR_SLOWHZ)

#   define  TCPREXMTTHRESH      3

/* -------------------------------------------------------------------- */

#   define  SEQ_LT(a,b)         ((int)((a)-(b)) < 0)
#   define  SEQ_LEQ(a,b)        ((int)((a)-(b)) <= 0)
#   define  SEQ_GT(a,b)         ((int)((a)-(b)) > 0)
#   define  SEQ_GEQ(a,b)        ((int)((a)-(b)) >= 0)

/* -------------------------------------------------------------------- */

```

```
#ifndef MAX
#   define  MAX(a,b)            (((a)>(b)) ? (a) : (b))
#endif
#ifndef MIN
#   define  MIN(a,b)            (((a)<(b)) ? (a) : (b))
#endif

/* -------------------------------------------------------------------- */

```

2.3.4.8.2. TCB

```

/* -------------------------------------------------------------------- */
/* $Id: tcp_tcb.c,v 1.3 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_tcb.c,v $
 * Revision 1.3  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.2  1995/10/10  08:15:17  mgream
 * cosmetic changes
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */


/* -------------------------------------------------------------------- */
/* Required Externals:
    QueueCreate
    QueueDestroy
 */

/* -------------------------------------------------------------------- */
/*  - - - TCB TABLE HANDLING - - -
 |
 |  The TCB table is where we keep a list of all the TCBs. This module
 |  instantiates the TCB table and provides a set of functions to
 |  create, destroy and lookup entries in the table for use with
 |  TCP processing. Note that the declaration of DECLARE_MAIN_VARIABLES
 |  refers to a single instantiation of the table, whereas otherwise
 |  only an external reference to the table is declared. There must be
 |  one and only one declaration of this parameter. Everything else
 |  can be static, there is no harm.
 */
/* -------------------------------------------------------------------- */
/* -------------------------------------------------------------------- */

/*  TcbEntry Table
 |
 |  The control block table is a global table that contains all the
 |  control blocks; one slot per entry that is allocated to callers.
 |  note that we either declare it, or declare it as an external
 |  reference, so that all the TCP modules can see it, and not see
 |  duplicates.
 */

#ifdef DECLARE_MAIN_VARIABLES
#   define  _SCOPE
#else
#   define  _SCOPE              extern
#endif

#   define  TCB_TABLE_SZ        1024

_SCOPE TcbEntry                 TcbTable[TCB_TABLE_SZ];
_SCOPE int                      TcbTableInitialised
#ifdef DECLARE_MAIN_VARIABLES
    = 0
#endif
;

/* -------------------------------------------------------------------- */

/*  TcbCreate
 |
 |  Create a new TCB entry by locating a free slot in the table, and
 |  thence initialisating the TCB. The index is returned back to the
 |  caller (and is expected to be used to set up a mapping into the
 |  table) with subsequent lookups and destroys.
 */
static int TcbCreate ()
  {
    int TcbIndex;

    if (TcbTableInitialised == 0)
      {
        TcbTableInitialised = 1;
        for (TcbIndex = 0; TcbIndex < TCB_TABLE_SZ; TcbIndex++)
            TcbTable[TcbIndex].Allocated = FALSE;
      }

    for (TcbIndex = 0; TcbIndex < TCB_TABLE_SZ &&
         TcbTable[TcbIndex].Allocated == TRUE; TcbIndex++)
        ;

    if (TcbIndex < TCB_TABLE_SZ)
      {
        TcbPtr Tcb = &TcbTable[TcbIndex];

        Tcb->Allocated = TRUE;
        Tcb->FragQueue = QueueCreate ();
        Tcb->_timer_ticks = 0;
      }

    return TcbIndex;
  }

/* -------------------------------------------------------------------- */

/*  TcbDestroy
 |
 |  There comes a time when Tcb's have to go back into the void, so
 |  here we have the function that shuts down the tcb and clears any
 |  internal memory that might be hanging around; including the
 |  fragment queue.
 */
static void TcbDestroy (TcbIndex)
    int TcbIndex;
  {
    if (TcbIndex >= 0 && TcbIndex < TCB_TABLE_SZ &&
         TcbTable[TcbIndex].Allocated == TRUE)
      {
        TcbPtr Tcb = &TcbTable[TcbIndex];

        QueueDestroy (Tcb->FragQueue);
        Tcb->Allocated = FALSE;
      }
  }

/* -------------------------------------------------------------------- */

/*  TcbLookup
 |
 |  Locate the TCB by simply applying the mapping; it was evisaged
 |  that there may be a message queue and some other miscellany
 |  in the table that would require us to perform initialisation
 |  here.
 */
static TcbPtr TcbLookup (TcbIndex)
    int TcbIndex;
  {
    if (TcbIndex >= 0 && TcbIndex < TCB_TABLE_SZ &&
        TcbTable[TcbIndex].Allocated == TRUE)
      {
        return &TcbTable[TcbIndex];
      }

    return NULL;
  }

/* -------------------------------------------------------------------- */

```

2.3.4.8.3. Init

```

/* -------------------------------------------------------------------- */
/* $Id: tcp_init.c,v 1.2 1995/10/10 08:15:17 mgream Exp $
 * $Log: tcp_init.c,v $
 * Revision 1.2  1995/10/10  08:15:17  mgream
 * cosmetic changes
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
/* Required Externals:
 */

/* -------------------------------------------------------------------- */
/*  - - - INIT PROCESSING - - -
 |
 |  Set up the TCB with all the default values.
 */
/* -------------------------------------------------------------------- */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */

/*  Init_Process
 |
 |  Initialise the TCB with all the required content, this includes
 |  the initial sequence numbers for both sender and receiver as well
 */
static void Init_Process (Tcb, isn)
    TcbPtr Tcb;
    tcp_seq isn;
  {

    Tcb->Flag_DelayedAck    = FALSE;
    Tcb->Flag_Ack           = FALSE;

    Tcb->Timer_Persist      = 0;
    Tcb->Timer_Retransmit   = 0;

    Tcb->snd_wnd            = (TCP_MAXWIN << TCP_MAX_WINSHIFT);
    Tcb->snd_una            = isn;
    Tcb->snd_nxt            = isn;
    Tcb->snd_wl1            = 0;
    Tcb->snd_wl2            = 0;
    Tcb->snd_max            = isn;

    Tcb->rcv_wnd            = (TCP_MAXWIN << TCP_MAX_WINSHIFT);
    Tcb->rcv_nxt            = isn;
    Tcb->rcv_adv            = isn;

    Tcb->last_ack_sent      = isn;

    Tcb->snd_cwnd           = (TCP_MAXWIN << TCP_MAX_WINSHIFT);
    Tcb->snd_ssthresh       = (TCP_MAXWIN << TCP_MAX_WINSHIFT);

    Tcb->t_idle             = 0;
    Tcb->t_rtt              = 0;
    Tcb->t_rtseq            = 0;
    Tcb->t_srtt             = TCPTV_SRTTBASE;
    Tcb->t_rttvar           = TCP_RTTDFLT * (PR_SLOWHZ << 2);
    Tcb->t_rttmin           = TCPTV_MIN;
    Tcb->max_sndwnd         = 0;

    Tcb->snd_scale          = 14;
    Tcb->rcv_scale          = 14;

    Tcb->Flag_Timestamp     = TRUE;
    Tcb->ts_recent          = 0;
    Tcb->ts_recent_age      = 0;

    Tcb->tcp_now            = 0;

    Tcb->t_rxtshift         = 0;
    Tcb->t_rxtcur           = Confine_Range (((TCPTV_SRTTBASE >> 2) +
                                            (TCPTV_SRTTDFLT << 2)) >> 1,
                                            TCPTV_MIN, TCPTV_REXMTMAX);
    Tcb->t_dupacks          = 0;
    Tcb->t_maxseg           = TCP_MSS;

  }

/* -------------------------------------------------------------------- */

```

2.3.4.8.4. Message

```

/* -------------------------------------------------------------------- */
/* $Id: tcp_msg.c,v 1.3 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_msg.c,v $
 * Revision 1.3  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.2  1995/10/10  08:15:17  mgream
 * cosmetic changes
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
/*  - - - MESSAGE ACCESS - - -
 |
 |  These primitives allow creation and destruction of internally
 |  represented TCP messages, along with converting between the
 |  internal and external (i.e. BONeS) representation.
 */

/* -------------------------------------------------------------------- */
/* -------------------------------------------------------------------- */

/*  MsgCreate
 |
 |  Create a msg by allocating off the heap.
 */
static MsgPtr MsgCreate ()
  {
    return (MsgPtr) __Balloc (sizeof (Message), "MsgCreate: Message");
  }

/* -------------------------------------------------------------------- */

/*  MsgDestroy
 |
 |  Return the message structure back to the heap by freeing it.
 */
static void MsgDestroy (Msg)
    MsgPtr Msg;
  {
    __Bfree ((void*)Msg);
  }

/* -------------------------------------------------------------------- */

/*  Field Handles
 |
 |  These are constant throughout module execution, so we can init
 |  them once, and leave it be after that.
 */
static  int             _fh_Initialised = 0;
static  field_handle    _fh_Length;
static  field_handle    _fh_Win;
static  field_handle    _fh_Seq;
static  field_handle    _fh_Ack;
static  field_handle    _fh_Flag_Ack;
static  field_handle    _fh_Flag_Timestamp;
static  field_handle    _fh_Time_Recent;
static  field_handle    _fh_Time_Now;
static  type_handle     _th_Msg_Transport_TCP;

/* -------------------------------------------------------------------- */

/*  _fh_Initialise
 |
 |  Initialise the field and type handles, we can used either Id's or
 |  ASCII names, i've preferred to the latter; there is no real
 |  significant difference.
 */
static void _fh_Initialise ()
  {
    if (_fh_Initialised != 0)
        return;
    _fh_Initialised = 1;
    _th_Msg_Transport_TCP = __GetTypeHandle ("Msg Transport TCP");
    _fh_Length = __GetFldHandle (_th_Msg_Transport_TCP, "Length");
    _fh_Win = __GetFldHandle (_th_Msg_Transport_TCP, "Win");
    _fh_Seq = __GetFldHandle (_th_Msg_Transport_TCP, "Seq");
    _fh_Ack = __GetFldHandle (_th_Msg_Transport_TCP, "Ack");
    _fh_Flag_Ack = __GetFldHandle (_th_Msg_Transport_TCP, "Flag_Ack");
    _fh_Flag_Timestamp = __GetFldHandle(_th_Msg_Transport_TCP,"Flag_Timestamp");
    _fh_Time_Recent = __GetFldHandle (_th_Msg_Transport_TCP, "Time_Recent");
    _fh_Time_Now = __GetFldHandle (_th_Msg_Transport_TCP, "Time_Now");
  }

/* -------------------------------------------------------------------- */

/*  MsgConvertFromBONeS
 |
 |  Convert a msg from the BONeS representation to one that we will
 |  use internally, which is much faster and acts as a point of
 |  isolation. Initialisation is carried out here, as well, if it
 |  hasn't already been done.
 */
static void MsgConvertFromBONeS (Msg, BMsg)
    MsgPtr Msg;
    arc_ptr BMsg;
  {
    if (_fh_Initialised == 0)
        _fh_Initialise ();
    Msg->Next = NULL;
    Msg->len = __GetINTEGERFldVal (BMsg, _fh_Length);
    Msg->win = __GetINTEGERFldVal (BMsg, _fh_Win);
    Msg->seq = __GetINTEGERFldVal (BMsg, _fh_Seq);
    Msg->ack = __GetINTEGERFldVal (BMsg, _fh_Ack);
    Msg->Flag_Ack = __GetINTEGERFldVal (BMsg, _fh_Flag_Ack);
    Msg->Flag_Timestamp = __GetINTEGERFldVal (BMsg, _fh_Flag_Timestamp);
    Msg->t_recent = __GetINTEGERFldVal (BMsg, _fh_Time_Recent);
    Msg->t_now = __GetINTEGERFldVal (BMsg, _fh_Time_Now);
  }

/* -------------------------------------------------------------------- */

/*  MsgConvertToBONeS
 |
 |  Convert a msg from the internal representation into the BONeS
 |  representation. Also make sure to initialise if it hasn't already
 |  been done.
 */
static void MsgConvertToBONeS (Msg, BMsg)
    MsgPtr Msg;
    arc_ptr BMsg;
  {
    if (_fh_Initialised == 0)
        _fh_Initialise ();
    __PutINTEGERFldVal (BMsg, _fh_Length, Msg->len);
    __PutINTEGERFldVal (BMsg, _fh_Win, Msg->win);
    __PutINTEGERFldVal (BMsg, _fh_Seq, Msg->seq);
    __PutINTEGERFldVal (BMsg, _fh_Ack, Msg->ack);
    __PutINTEGERFldVal (BMsg, _fh_Flag_Ack, Msg->Flag_Ack);
    __PutINTEGERFldVal (BMsg, _fh_Flag_Timestamp, Msg->Flag_Timestamp);
    __PutINTEGERFldVal (BMsg, _fh_Time_Recent, Msg->t_recent);
    __PutINTEGERFldVal (BMsg, _fh_Time_Now, Msg->t_now);
  }

/* -------------------------------------------------------------------- */

```

2.3.4.8.5. Outgoing Queue

```

/* -------------------------------------------------------------------- */
/* $Id: tcp_outqueue.c,v 1.3 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_outqueue.c,v $
 * Revision 1.3  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.2  1995/10/10  08:15:17  mgream
 * cosmetic changes
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
/* Required Externals:
    MsgCreate
    MsgDestroy
    MsgConvertToBONeS
 */

/* -------------------------------------------------------------------- */
/*  - - - OUTPUT QUEUE - - -
 |
 |  The output queue is used for the storage of TCP messages that
 |  are to be transmitted during a run of a tcp entity; we do this
 |  because it is possible for multiple messages to be sent at once,
 |  so we need to queue them and then convert them into a vector
 |  which is then used in the BONeS environment to fan out to lots
 |  of individual messages. This Queue is a FIFO.
 |
 */
/* -------------------------------------------------------------------- */

static MsgPtr OutQueue_Head = NULL;

/* -------------------------------------------------------------------- */

/*  OutQueue_Initialise
 |
 |  Initialise the output queue; note that it is safe to do it using
 |  static variables like this because we will never have concurrent
 |  instances of the output queue occuring (i.e. once we enter the TCP
 |  primitive, we never go back to the BONeS environment until we
 |  finish, at which point we extract everything).
 */
static void OutQueue_Initialise (void)
  {
    OutQueue_Head = NULL;
  }

/* -------------------------------------------------------------------- */

/*  OutQueue_EnQueue
 |
 |  EnQueue a message onto the Output Queue. This appends the passed
 |  message to the end of the queue.
 */
static void OutQueue_EnQueue (Msg)
    MsgPtr Msg;
  {

    if (OutQueue_Head == NULL)
      {
        OutQueue_Head = Msg;
      }
    else
      {
        MsgPtr MsgTmp = OutQueue_Head;

        while (MsgTmp->Next != NULL)
            MsgTmp = MsgTmp->Next;

        MsgTmp->Next = Msg;
      }
```

```c

    Msg->Next = NULL;
  }

/* -------------------------------------------------------------------- */

/*  OutQueue_DeQueue
 |
 |  DeQueue a message from the Output Queue.
 */
static MsgPtr OutQueue_DeQueue ()
  {
    MsgPtr Msg = OutQueue_Head;

    if (OutQueue_Head != NULL)
        OutQueue_Head = OutQueue_Head->Next;

    return Msg;
  }

/* -------------------------------------------------------------------- */

/*  OutQueue_GetSize
 |
 |  Sometimes, size does matter, so we do need to know!
 */
static int OutQueue_GetSize ()
  {
    MsgPtr Msg;
    int Size = 0;

    for (Msg = OutQueue_Head; Msg != NULL; Msg = Msg->Next)
        Size++;

    return Size;
  }

/* -------------------------------------------------------------------- */

/*  OutQueue_Extract
 |
 |  We process the content of the Queue by extracting each message
 |  in succession and adding into an outgoing vector. Note that this
 |  also includes having to set up the vector, and then put each
 |  element into it. There is a conversion mechanism that will map
 |  from the Internal representation into the BONeS representation.
 */
static void OutQueue_Extract (MsgVector)
    arc_ptr MsgVector;
  {
    MsgPtr Msg;
    arc_t MsgBONeS;
    int Index;
    int Size;
    type_handle TypeHandle;

    Size = OutQueue_GetSize ();
    if (Size == 0)
        return;

    MsgBONeS.enable = 0;
    TypeHandle = __GetTypeHandle ("Msg Transport TCP");
    __CreateCOMPOSITESubType (TypeHandle, &MsgBONeS);
    __CreateVECTOR (TypeHandle, Size, &MsgBONeS, MsgVector);
    __FreeArc (&MsgBONeS);

    for (Index = 0; Index < Size; Index++)
      {
        Msg = OutQueue_DeQueue ();
        MsgConvertToBONeS (Msg, &MsgBONeS);
        MsgDestroy (Msg);

        __PutVECTORElmVal (MsgVector, Index, &MsgBONeS);
        __FreeArc (&MsgBONeS);
      }

  }

/* -------------------------------------------------------------------- */

```

2.3.4.8.6. Fragment

```

/* -------------------------------------------------------------------- */
/* $Id: tcp_frag.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_frag.c,v $
 * Revision 1.2  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/*
 |  --- Fragment ---
 |
 |  Module:
 |      Fragment Reassembly Module
 |  Author:
 |      Matthew Gream
 |  Description:
 |      The module implements primitives for maintaining a chain of
 |      fragments with the ability to add and remove fragments from
 |      the chain; specifically this caters for TCP fragments in a
 |      manner similar to TCP_REASS from BSD Net/3.
 |  Date:
 |      September 1995
 |  Revision:
 |      $Id: tcp_frag.c,v 1.2 1995/12/21 11:08:30 mgream Exp $
 */

/* -------------------------------------------------------------------- */

#ifdef TEST

#   include <stdio.h>
#   include <stdlib.h>
#   include <string.h>
#   include <assert.h>

#   define  _MALLOC(s,id)       malloc (s)
#   define  _FREE(p)            free (p)

#else

#   define  assert(x)
#   define  _MALLOC(s,id)       __Balloc (s, id)
#   define  _FREE(p)            __Bfree ((void *)p)

#endif

/* -------------------------------------------------------------------- */

#ifndef _SEQ_GT
#   define  _SEQ_GT(a,b)            ((int)((a) - (b)) > 0)
#endif

/* -------------------------------------------------------------------- */
typedef unsigned long SeqNo;

/* -------------------------------------------------------------------- */
/* Fragment :: Maintain information about one single Fragment */
typedef struct Fragment_ST {
    struct Fragment_ST * Next;                /* Next Fragment in chain */
    struct Fragment_ST * Prev;            /* Previous Fragment in chain */
    SeqNo Sequence;                 /* Sequence number of this Fragment */
    SeqNo Length;                            /* Length of this Fragment */
  } Fragment;

/* -------------------------------------------------------------------- */
/* Queue :: Maintain information about the Queue of Fragments */
typedef struct Queue_ST {
    Fragment * Head;                      /* Head of the Fragment chain */
  } Queue;
```

```

/* -------------------------------------------------------------------- */
#ifndef P
#ifdef ANSIC
#   define  P(x)                x
#else
#   define  P(x)                ()
#endif
#endif

static Fragment * FragmentCreate P ((SeqNo Sequence, SeqNo Length));
static void FragmentDestroy P ((Fragment * Frag));
static Fragment * FragmentRemove P ((Fragment * Head, Fragment * Frag));
static Fragment * FragmentInsert P ((Fragment * Head, Fragment * PFrag,
                Fragment * Frag));
static Queue * QueueCreate P ((void));
static void QueueDestroy P ((Queue * Que));
static void QueueClear P ((Queue * Que));
static void QueueAddFragment P ((Queue * Que, SeqNo Sequence, SeqNo Length));
static SeqNo QueueGetHeadSequence P ((Queue * Que));
static SeqNo QueueGetHeadLength P ((Queue * Que));

/* -------------------------------------------------------------------- */
/* FragmentCreate:
 |  Inputs:
 |      SeqNo Sequence          -- Sequence number of the Fragment
 |      SeqNo Length            -- Length of the Fragment
 |  Outputs:
 |      Fragment *              -- The created Fragment
 |  Description:
 |      A new Fragment entry is created with the specified parameters
 |      (Sequence and Length).
 */
static Fragment * FragmentCreate (Sequence, Length)
    SeqNo Sequence;
    SeqNo Length;
  {
    Fragment * Frag = (Fragment *) _MALLOC (sizeof (Fragment), "Fragment");
    assert (Frag != NULL);
    Frag->Next = NULL;
    Frag->Prev = NULL;
    Frag->Sequence = Sequence;
    Frag->Length = Length;
    return Frag;
  }

/* -------------------------------------------------------------------- */
/* FragmentDestroy:
 |  Inputs:
 |      Fragment * Frag         -- Fragment we want to destroy
 |  Outputs:
 |      n/a
 |  Description:
 |      The Fragment is removed from existance.
 */
static void FragmentDestroy (Frag)
    Fragment * Frag;
  {
    assert (Frag != NULL);
    _FREE (Frag);
  }

/* -------------------------------------------------------------------- */
/* FragmentRemove:
 |  Inputs:
 |      Fragment * Head         -- The Head of the Fragment chain
 |      Fragment * Frag         -- The Fragment to remove
 |  Outputs:
 |      Fragment *              -- The new Head of the Fragment chain
 |  Description:
 |      The specified fragment entry is removed from the chain, with
 |      the assurance that the head of the chain is updated to reflect
 |      a new value should it change.
 */
static Fragment * FragmentRemove (Head, Frag)
    Fragment * Head;
    Fragment * Frag;
  {
    assert (Head != NULL);
    assert (Frag != NULL);
    if (Frag->Prev != NULL)
```

```
        Frag->Prev->Next = Frag->Next;
    else
        Head = Frag->Next;
    if (Frag->Next != NULL)
        Frag->Next->Prev = Frag->Prev;
    FragmentDestroy (Frag);
    return Head;
  }

/* -------------------------------------------------------------------- */
/* FragmentInsert:
 |  Inputs:
 |      Fragment * Head         -- The Head of the Fragment chain
 |      Fragment * PFrag        -- The Fragment previous to the slot
 |                                 that we want to insert
 |      Fragment * Frag         -- The Fragment to be inserted
 |  Outputs:
 |      Fragment *              -- The new Head of the Fragment chain
 |  Description:
 |      The new Fragment is inserted after the specified Previous
 |      Fragment with all appropriate link pointers updated to
 |      reflect the insertion. The head value is altered and returned
 |      if it changes.
 */
static Fragment * FragmentInsert (Head, PFrag, Frag)
    Fragment * Head;
    Fragment * PFrag;
    Fragment * Frag;
  {
    assert (Frag != NULL);
    if (Head == NULL)
      {
        Frag->Next = NULL;
        Frag->Prev = NULL;
        return Frag;
      }
    else if (PFrag == NULL)
      {
        Frag->Next = Head;
        Frag->Prev = NULL;
        if (Head != NULL)
            Head->Prev = Frag;
        Head = Frag;
      }
    else
      {
        Frag->Prev = PFrag;
        Frag->Next = PFrag->Next;
        if (PFrag->Next != NULL)
            PFrag->Next->Prev = Frag;
        PFrag->Next = Frag;
      }
    return Head;
  }

/* -------------------------------------------------------------------- */
/* QueueCreate:
 |  Inputs:
 |      n/a
 |  Outputs:
 |      Queue *                 -- The created Queue
 |  Description:
 |      A new Queue is created and initialised.
 */
static Queue * QueueCreate ()
  {
    Queue * Que = (Queue *) _MALLOC (sizeof (Queue), "Queue Context");
    assert (Que != NULL);
    Que->Head = NULL; /* XXX: splodes if assert failed anyway */
    return Que;
  }

/* -------------------------------------------------------------------- */
/* QueueDestroy:
 |  Inputs:
 |      Queue * Que             -- The Queue to destroy
 |  Outputs:
 |      n/a
 |  Description:
 |      The specified Queue is destroyed; including any internal
 |      elements that may still be present.
 */
static void QueueDestroy (Que)
    Queue * Que;
  {
    assert (Que != NULL);
    QueueClear (Que);
    _FREE (Que);
  }

/* --------------------------------------------------------------------- */
/* QueueClear:
 |  Inputs:
 |      Queue * Que             -- The Queue to clear
 |  Outputs:
 |      n/a
 |  Description:
 |      All internal elements are cleared from the Queue.
 */
static void QueueClear (Que)
    Queue * Que;
  {
    while (Que->Head != NULL)
        Que->Head = FragmentRemove (Que->Head, Que->Head);
  }

/* --------------------------------------------------------------------- */
/* QueueAddFragment:
 |  Inputs:
 |      Queue * Que             -- The Queue to add into.
 |      SeqNo Sequence          -- Sequence Number of the Fragment.
 |      SeqNo Length            -- Length of the Fragment.
 |  Outputs:
 |      n/a
 |  Description:
 |      The incoming fragment will be trimmed and added into its
 |      appropriate position in the queue; which may require that
 |      other entries are also trimmed. In addition, it could occur
 |      that this or other elements are wholey removed.
 */
static void QueueAddFragment (Que, Sequence, Length)
    Queue * Que;
    SeqNo Sequence;
    SeqNo Length;
  {
    Fragment * Frag = FragmentCreate (Sequence, Length);
    Fragment * QFrag;
    Fragment * TFrag;
    Fragment * EFrag;
    int Range;

#ifdef TEST
    printf ("QAF: Inserting Fragment : (%lu, %lu)\n", Sequence, Length);
#endif

    if (Que->Head == NULL)
      {
        Que->Head = FragmentInsert (NULL, NULL, Frag);
        return;
      }

    /* Locate the slot for our Fragment.
     */
    QFrag = Que->Head;
    while (QFrag != NULL) {
        if (_SEQ_GT (QFrag->Sequence, Frag->Sequence))
            break;
        EFrag = QFrag;
        QFrag = QFrag->Next;
      }

#ifdef TEST
    if (QFrag == NULL)
        printf ("   : Slot located = (%lu, %lu) AFTER\n",
                EFrag->Sequence, EFrag->Length);
    else
        printf ("   : Slot located = (%lu, %lu) BEFORE\n",
                QFrag->Sequence, QFrag->Length);
#endif

    /* Trim the front of the Fragment depending the previous
       entry in the Fragment chain
     */
    if (QFrag == NULL || QFrag->Prev != NULL) {
        QFrag = (QFrag == NULL) ? EFrag : QFrag->Prev;
        Range = (int)(QFrag->Sequence + QFrag->Length - Frag->Sequence);
        if (Range > 0) {
            if (Range >= Frag->Length) {
                FragmentDestroy (Frag);
#ifdef TEST
                printf ("   : Fragment completely covered by (%lu, %lu)\n",
                        QFrag->Sequence, QFrag->Length);
#endif
                return;
              }
            Frag->Sequence += Range;
            Frag->Length -= Range;
#ifdef TEST
            printf ("   : Trimmed Fragment = (%lu, %lu)\n", Frag->Sequence,
                       Frag->Length);
#endif
          }
        QFrag = QFrag->Next;
      }

    /* Remove or Trim subsequent Fragments in the Fragment chain
     */
    while (QFrag != NULL) {
        Range = (int)((Frag->Sequence + Frag->Length) - QFrag->Sequence);
        if (Range <= 0)
            break;
        if (Range < QFrag->Length) {
#ifdef TEST
            printf ("   : Trimming entry (%lu, %lu) to (%lu, %lu)\n",
                        QFrag->Sequence, QFrag->Length,
                        QFrag->Sequence + Range, QFrag->Length - Range);
#endif
            QFrag->Sequence += Range;
            QFrag->Length -= Range;
            break;
          }
#ifdef TEST
        printf ("   : Removing entry (%lu, %lu) as obsolete\n",
                        QFrag->Sequence, QFrag->Length);
#endif
        TFrag = QFrag;
        QFrag = QFrag->Next;
        Que->Head = FragmentRemove (Que->Head, TFrag);
      }
#ifdef TEST
    if (QFrag == NULL)
        printf ("   : Inserting after entry (%lu, %lu)\n",
                        EFrag->Sequence, EFrag->Length);
    else if (QFrag->Prev == NULL)
        printf ("   : Inserting at Head\n");
    else
        printf ("   : Inserting after entry (%lu, %lu)\n",
                        QFrag->Prev->Sequence, QFrag->Prev->Length);
#endif
    Que->Head = FragmentInsert (Que->Head, (QFrag == NULL) ?
                    EFrag : QFrag->Prev, Frag);

#ifdef TEST
    printf ("   : Fragment added; Queue contents:\n");
    printf ("   : ");
    for (Frag = Que->Head; Frag != NULL; Frag = Frag->Next)
        printf ("(%lu, %lu) ", Frag->Sequence, Frag->Length);
    printf ("\n");
#endif
  }

#ifdef UNUSED_CODE
/* --------------------------------------------------------------------- */
/* QueueGetHeadSequence:
 |  Inputs:
 |      Queue * Que             -- The Queue that we are working on.
 |  Outputs:
 |      SeqNo                   -- Sequence number of the head element.
 |  Description:
 |      Returns the Sequence number of the very top element on the
 |      Queue, noting that it is assumed that there is an actual
 |      element at the head of the Queue.
 */
```

```
static SeqNo QueueGetHeadSequence (Que)
    Queue * Que;
  {
    assert (Que->Head != NULL);
    return Que->Head->Sequence;
  }
#endif

/* ------------------------------------------------------------------- */
/* QueueGetSize
 |  Inputs:
 |      Queue * Que             -- The Queue we are querying.
 |  Outputs:
 |      int                     -- Number of Fragments on the Queue.
 |  Description:
 |      The number of fragments that exist on the reassembly queue
 |      are counted and returned.
 */
static int QueueGetSize (Que)
    Queue * Que;
  {
    Fragment * Frag;
    int QueSz = 0;
    for (Frag = Que->Head; Frag != NULL; Frag = Frag->Next)
        QueSz++;
    return QueSz;
  }

/* ------------------------------------------------------------------- */
/* QueueGetHeadLength
 |  Inputs:
 |      Queue * Que             -- The Queue we are working on.
 |  Outputs:
 |      SeqNo                   -- Total Length of top Fragment.
 |  Description:
 |      The maximal contiguous sequence range from the first
 |      fragment in the Queue is returned.
 */
static SeqNo QueueGetHeadLength (Que)
    Queue * Que;
  {
    SeqNo Length = 0;
    if (Que->Head != NULL) {
        SeqNo SequenceNext;
        do {
            Length += Que->Head->Length;
            SequenceNext = Que->Head->Sequence + Que->Head->Length;
            Que->Head = FragmentRemove (Que->Head, Que->Head);
          } while (Que->Head != NULL && SequenceNext == Que->Head->Sequence);
      }
#ifdef TEST
    { Fragment * Frag;
    printf ("QGHL: Length Extracted (%lu); Queue contents:\n", Length);
    printf ("    : ");
    for (Frag = Que->Head; Frag != NULL; Frag = Frag->Next)
        printf ("(%lu, %lu) ", Frag->Sequence, Frag->Length);
    printf ("\n");
    }
#endif
    return Length;
  }

/* ------------------------------------------------------------------- */

#ifdef TEST
void main (argc, argv)
    int argc;
    char ** argv;
  {
    Queue * Que = QueueCreate ();
    QueueAddFragment (Que, 10, 10);
    QueueAddFragment (Que, 40, 10);
    QueueAddFragment (Que, 30, 05);
    QueueAddFragment (Que, 15, 10);
    QueueAddFragment (Que, 31, 01);
    QueueAddFragment (Que, 50, 02);
    QueueGetHeadLength (Que);
    QueueAddFragment (Que, 15, 30);
    QueueGetHeadLength (Que);
    QueueDestroy (Que);
  }
```

```
#endif

/* ---------------------------------------------------------------------- */

```

2.3.4.8.7. Input

```

/* ---------------------------------------------------------------------- */
/* $Id: tcp_input.c,v 1.3 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_input.c,v $
 * Revision 1.3  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.2  1995/10/10  08:15:17  mgream
 * cosmetic changes
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* ---------------------------------------------------------------------- */

/* ---------------------------------------------------------------------- */
/* Required Externals:
    _BONeS_Get_Send_Buffer_Sz
    _BONeS_Set_Send_Buffer_Sz
    _BONeS_Get_Recv_Buffer_Sz
    _BONeS_Set_Recv_Buffer_Sz
    QueueAddFragment
    QueueGetHeadLength
 */

/* ---------------------------------------------------------------------- */
/*    - - - INPUT PROCESSING - - -
 |
 |  Input processing occurs which a received segment, what we do is
 |  perform a series of steps which are abstracted down into two
 |  phases, the first phase is a verification of the message including
 |  alterations to it if need be; the second is actually processing
 |  the content of the message.
 */
/* ---------------------------------------------------------------------- */
/* ---------------------------------------------------------------------- */

/*  EXIT_FLAGS
 |
 |  These flags are used as function returns to indicate appropriate
 |  action.
 */
#   define  I_STOP           0
#   define  I_NEXT           1
#   define  I_SKIP           2

/* ---------------------------------------------------------------------- */

/*  Input_Info_ST
 |
 |  This data structure is used to contain local information used
 |  during the Input phase; it was considered better to do it this
 |  way rather than pass a lot of variables through subroutines.
 */
typedef struct Input_Info_ST
  {
    boolean needoutput;
    int acked;
  } Input_Info;
typedef Input_Info * InPtr;

/* ---------------------------------------------------------------------- */

/*  In_Update_Receive_Window
 |
 |  Update the receive window.
 */
static int In_Update_Receive_Window (Tcb, Msg, inp)
```

```
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    Tcb->rcv_wnd = MAX (TCP_MAXWIN << TCP_MAX_WINSHIFT,
            Tcb->rcv_adv - Tcb->rcv_nxt);

    return I_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  In_Check_Segment_Position
 |
 |  Check the segments position within our receive window; we do this
 |  by looking to see how much of is outside the receive window, and
 |  if all of it is, then we throw the message away and send out an
 |  ack.
 */
static int In_Check_Segment_Position (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    int todrop;

    todrop = Tcb->rcv_nxt - Msg->seq;

    if (todrop > 0)
      {
        if (todrop >= Msg->len)
          {
            Tcb->Flag_Ack = TRUE;

            todrop = Msg->len;
          }

        Msg->len = Msg->len - todrop;
        Msg->seq = Msg->seq + todrop;
      }

    return I_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  In_Trim_Segment_Content
 |
 |  What we do here is check to see how much of the segment lies outside
 |  of the window; and we attempt to throw away that which does. If the
 |  segment is fully outside, then drop the message and throw back an
 |  ack to our peer to indicate so.
 */
static int In_Trim_Segment_Content (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    int todrop;

    todrop = (Msg->seq + Msg->len) - (Tcb->rcv_nxt + Tcb->rcv_wnd);

    if (todrop > 0)
      {
        if (todrop >= Msg->len)
          {
            Tcb->Flag_Ack = TRUE;

            if (!(Tcb->rcv_wnd == 0 && Msg->seq == Tcb->rcv_nxt))
              {
                Output_Process (Tcb, FALSE);

                return I_STOP;
             }
          }

        Msg->len = Msg->len - todrop;
      }

    return I_NEXT;
  }
```

```

/* ------------------------------------------------------------------- */

/*  In_Process_Timestamp
 |
 |  Extract the timestamp and related information from the message, we
 |  make sure that we only accept timestamps that are from valid
 |  messages, and not retransmits.
 */
static int In_Process_Timestamp (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    if (Msg->Flag_Timestamp == TRUE &&
        SEQ_LEQ (Msg->seq, Tcb->last_ack_sent) &&
        SEQ_LT (Tcb->last_ack_sent, Msg->seq + Msg->len))
      {
        Tcb->ts_recent_age = Tcb->tcp_now;
        Tcb->ts_recent = Msg->t_now;
      }

    return I_NEXT;
  }

/* ------------------------------------------------------------------- */

/*  In_Initial_Processing
 |
 |  The input stage requires some initial processing, this takes the
 |  form of carrying out several validity checks on the segment, and
 |  possibly tossing away the segment if we happen to need to do so.
 |  The processing we do is as follows:
 |  1. Receive Window -- recompute the receive window (this is not
 |      affected by the incoming message, but we only need to have
 |      it done for input processing).
 |  2. Segment Position -- check the segments position in the
 |      receive window, as we may need to drop it.
 |  3. Trim Segment -- cut out upper and lower chunks from the
 |      segment if they fall outside the window, note that this
 |      may also cause the entire segment to be dropped.
 |  4. Process Timestamp -- extract the timestamp option from the
 |      segment and update local fields.
 |  The first thing we do, also, is to make sure that we indicate that
 |  we are not idle anymore.
 */
static int In_Initial_Process (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    Tcb->t_idle = 0;

    if (In_Update_Receive_Window (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Check_Segment_Position (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Trim_Segment_Content (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Process_Timestamp (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    return I_NEXT;
  }

/* ------------------------------------------------------------------- */

/*  In_Ack_Duplicate_Acks
 |
 |  This is where we process duplicate acks. We increase them until a
 |  threshold is reached, at which point we scale back the slow start
 |  threshold and the congestion window, then fire off tcp output as
 |  a guess that we seen a packet dropped (but not hit the retransmit
 |  threshold). If we are more than the threshold of duplicate acks,
 |  we pump up the congestion window by a segment so as to keep the
 |  pipe full : and kick output processing.
 */
static int In_Ack_Duplicate_Acks (Tcb, Msg, inp)
```

```
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    if (SEQ_LEQ (Msg->ack, Tcb->snd_una))
      {
        if (Msg->len == 0 && (Msg->win << Tcb->snd_scale) == Tcb->snd_wnd)
          {
            if (Tcb->Timer_Retransmit == 0 || Msg->ack != Tcb->snd_una)
              {
                Tcb->t_dupacks = 0;
              }
            else if (++Tcb->t_dupacks == TCPREXMTTHRESH)
              {
                tcp_seq onxt;
                u_int win;

                onxt = Tcb->snd_nxt;

                win = MIN (Tcb->snd_wnd, Tcb->snd_cwnd) / 2 / Tcb->t_maxseg;
                if (win < 2)
                    win = 2;

                Tcb->snd_ssthresh = win * Tcb->t_maxseg;
                Tcb->Timer_Retransmit = 0;
                Tcb->t_rtt = 0;
                Tcb->snd_nxt = Msg->ack;
                Tcb->snd_cwnd = Tcb->t_maxseg;

                Output_Process (Tcb, FALSE);

                Tcb->snd_cwnd = Tcb->snd_ssthresh +
                    Tcb->t_maxseg * Tcb->t_dupacks;

                if (SEQ_GT (onxt, Tcb->snd_nxt))
                  {
                    Tcb->snd_nxt = onxt;
                  }

                return I_STOP;
             }
            else if (Tcb->t_dupacks > TCPREXMTTHRESH)
              {
                Tcb->snd_cwnd = Tcb->snd_cwnd + Tcb->t_maxseg;

                Output_Process (Tcb, FALSE);

                return I_STOP;
              }
          }
        else
          {
            Tcb->t_dupacks = 0;
          }

        return I_SKIP;
      }

    return I_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  Process_Transmit_Timer
 |
 |  Compute a new smoothed RTT value.
 */
static void Process_Transmit_Timer (Tcb, rtt)
    TcbPtr Tcb;
    int rtt;
  {
    if (Tcb->t_srtt != 0)
      {
        short delta;

        delta = rtt - 1 - (Tcb->t_srtt >> TCP_RTT_SHIFT);

        Tcb->t_srtt = Tcb->t_srtt + delta;
        if (Tcb->t_srtt <= 0)
          {
            Tcb->t_srtt = 1;
          }

        if (delta < 0)
          {
            delta = -delta;
          }

        delta = delta - (Tcb->t_rttvar >> TCP_RTTVAR_SHIFT);

        Tcb->t_rttvar = Tcb->t_rttvar - delta;
        if (Tcb->t_rttvar <= 0)
          {
            Tcb->t_rttvar = 1;
          }
      }
    else
      {
        Tcb->t_srtt = rtt << TCP_RTT_SHIFT;
        Tcb->t_rttvar = rtt << (TCP_RTTVAR_SHIFT - 1);
      }

    Tcb->t_rtt = 0;
    Tcb->t_rxtshift = 0;
    Tcb->t_rxtcur = Confine_Range (Get_Retransmit_Value (),
                                  Tcb->t_rttmin, TCPTV_REXMTMAX);
  }

/* -------------------------------------------------------------------- */

/*  In_Ack_Update_Round_Trip_Time
 |
 |  Update our round trip time estimators, taking into account two
 |  cases, the first being where we have a timestamp, so we can use
 |  this (much more reliable) information to do the RTT. Otherwise,
 |  if the ack is greater than that which we sent out to time for
 |  this segment, then we use our estimated rtt time.
 |  CHECK THIS.
 */
static int In_Ack_Update_Round_Trip_Time (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    if (Msg->Flag_Timestamp == TRUE)
      {
        Process_Transmit_Timer (Tcb, Tcb->tcp_now - Msg->t_recent + 1);
      }
    else if (Tcb->t_rtt != 0 && SEQ_GT (Msg->ack, Tcb->t_rtseq))
      {
        Process_Transmit_Timer (Tcb, Tcb->t_rtt);
      }

    return I_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  In_Ack_Update_Retransmit_Timer
 |
 |  The retransmit timer needs to be either stopped or restarted
 |  depending on two conditions; the first is the case where we have
 |  been acked up to everything we have sent; which means that we
 |  don't need to be retransmitting. Alternatively, if we are not
 |  persisting, then do go in for the retransmit.
 |  CHECK THIS.
 */
static int In_Ack_Update_Retransmit_Timer (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    if (Msg->ack == Tcb->snd_max)
      {
        Tcb->Timer_Retransmit = 0;
        inp->needoutput = TRUE;
      }
    else if (Tcb->Timer_Persist == 0)
      {
        Tcb->Timer_Retransmit = Tcb->t_rxtcur;
      }

    return I_NEXT;
  }

/* ------------------------------------------------------------------- */

/*  In_Ack_Update_Congestion
 |
 |  Update the congestion window, what we do is increase it just a tad
 |  but constrain it to the maximum window we can send.
 */
static int In_Ack_Update_Congestion (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    int cw;
    int incr;

    cw = Tcb->snd_cwnd;

    incr = Tcb->t_maxseg;

    if (cw > Tcb->snd_ssthresh)
      {
        incr = incr * incr / cw;
      }

    Tcb->snd_cwnd = MIN (cw + incr, TCP_MAXWIN << Tcb->snd_scale);

    return I_NEXT;
  }

/* ------------------------------------------------------------------- */

/*  In_Ack_Process_Ack
 |
 |  Here, the ACK is actually used to slop out data from the transmit
 |  buffer; what we do is look at how much has been acked, and it
 |  either covers the entire buffer, or only part thereof. Note that
 |  in TCP, we don't have selective acks, which kind of makes this
 |  process easier (at the cost of performance :-). Having finished
 |  updating the buffer, we update the next and unacknowledged
 |  sequence number fields in the Tcb.
 */
static int In_Ack_Process_Ack (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    int buffer_sz = _BONeS_Get_Send_Buffer_Sz (Tcb);

    if (inp->acked > buffer_sz)
      {
        _BONeS_Set_Send_Buffer_Sz (Tcb, 0);
        Tcb->snd_wnd = Tcb->snd_wnd - buffer_sz;
      }
    else
      {
        _BONeS_Set_Send_Buffer_Sz (Tcb, buffer_sz - inp->acked);
        Tcb->snd_wnd = Tcb->snd_wnd - inp->acked;
      }

    Tcb->snd_una = Msg->ack;

    if (SEQ_LT (Tcb->snd_nxt, Tcb->snd_una))
      {
        Tcb->snd_nxt = Tcb->snd_una;
      }

    return I_NEXT;
  }

/* ------------------------------------------------------------------- */

/*  In_Ack_Update_Remote
 |
 |  If we have a pile of duplicate acks, then we may need to scale back
 |  the congestion window to the slow start threshold. Also, what we do
 |  is drop out here if we are being acked for data that is above our
 |  window (should neeever happen ...).
 */
```

```
static int In_Ack_Update_Remote (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    if (Tcb->t_dupacks > TCPREXMTTHRESH && Tcb->snd_cwnd > Tcb->snd_ssthresh)
      {
        Tcb->snd_cwnd = Tcb->snd_ssthresh;
      }

    Tcb->t_dupacks = 0;

    if (SEQ_GT (Msg->ack, Tcb->snd_max))
      {
        Tcb->Flag_Ack = TRUE;

        Output_Process (Tcb, FALSE);

        return I_STOP;
      }

    inp->acked = Msg->ack - Tcb->snd_una;

    return I_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  In_Ack_Process
 |
 |  We must process lots of things in the input message relating to
 |  messages when they have acks on them. The following is what we
 |  need to look at:
 |  1. Duplicate Acks -- These fire up the "fast retransmit" mechanism
 |      of TCP that assumes that 3 duplicate acks are a sign of
 |      lost segments.
 |  2. Update Remote -- Check to see how much data is acked, and
 |      more fundamentally, whether or not the ack is within our
 |      window.
 |  3. Update Round Trip Time -- This ack may be coming back from
 |      a segment we were timing, or alternatively we may use what
 |      was in the timestamp.
 |  4. Stop Retransmit Timer -- Stop or continue the retransmit
 |      timer depending on whether this segment is in the window.
 |  5. Update Congestion -- Must update the congestion window
 |      based on the incoming acks ("Ack clocking").
 |  6. Process Ack -- Finally, the ack is processed so that we
 |      release transmit buffer content and update the appropriate
 |      sequence numbers.
 */
static int In_Ack_Process (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    switch (In_Ack_Duplicate_Acks (Tcb, Msg, inp))
      {
        case I_SKIP:
            return I_SKIP;

        case I_STOP:
            return I_STOP;

        case I_NEXT:
            break;
      }

    if (In_Ack_Update_Remote (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Ack_Update_Round_Trip_Time (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Ack_Update_Retransmit_Timer (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Ack_Update_Congestion (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Ack_Process_Ack (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    return I_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  In_Window_Update
 |
 |  Process for a window update, by looking at the sent sequence numbers
 |  and the updated window. What we are trying to do is make sure that
 |  we only process window updates on acks where the update is not an
 |  old one!
 */
static int In_Window_Update (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    if (Msg->Flag_Ack == TRUE && (SEQ_LT (Tcb->snd_wl1, Msg->seq) ||
            (Tcb->snd_wl1 == Msg->seq && (SEQ_LT (Tcb->snd_wl2, Msg->ack) ||
            (Tcb->snd_wl2 == Msg->ack &&
            (Msg->win << Tcb->snd_scale) > Tcb->snd_wnd)))))
      {
        Tcb->snd_wnd = (Msg->win << Tcb->snd_scale);
        Tcb->snd_wl1 = Msg->seq;
        Tcb->snd_wl2 = Msg->ack;

        if (Tcb->snd_wnd > Tcb->max_sndwnd)
          {
            Tcb->max_sndwnd = Tcb->snd_wnd;
          }

        inp->needoutput = TRUE;
      }

    return I_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  In_Data_Process
 |
 |  Here we process the data that is in the segment, there are two
 |  cases (only for purposes of optimisation); the first is where
 |  we are receiving the next segment of data inline and there is
 |  nothing on the queue. We can accept the data straight away and
 |  pass it up to the application. The second case is where we do
 |  have existing fragments, so we stick this into the reassembly
 |  queue and immediately attempt to extract anything that is at
 |  the head of the queue. We setup a delayed ack flag for the
 |  inline case, and a normal ack for the other.
 */
static int In_Data_Process (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    int len;
    int buffer_sz = _BONeS_Get_Recv_Buffer_Sz (Tcb);

    if (Msg->len > 0)
      {
        if (Msg->seq == Tcb->rcv_nxt && QueueGetSize (Tcb->FragQueue) == 0)
          {
            Tcb->Flag_DelayedAck = TRUE;
            Tcb->rcv_nxt = Tcb->rcv_nxt + Msg->len;

            _BONeS_Set_Recv_Buffer_Sz (Tcb, buffer_sz + Msg->len);
          }
        else
          {
            QueueAddFragment (Tcb->FragQueue, Msg->seq, Msg->len);
            len = QueueGetHeadLength (Tcb->FragQueue);
            if (len > 0)
              {
                Tcb->rcv_nxt = Tcb->rcv_nxt + len;
                _BONeS_Set_Recv_Buffer_Sz (Tcb, buffer_sz + len);
              }

            Tcb->Flag_Ack = TRUE;
          }
      }

    return I_NEXT;
  }

/* ------------------------------------------------------------------ */

/*  In_Content_Process
 |
 |  Process the content of a message, taking several steps. These are
 |  the things that need to be done:
 |  1. Ack Processing -- do all the things that occur when we get
 |      messages with the ack bit set.
 |  2. Window Updating -- update the receive window.
 |  3. Data Processing -- extract the content of the message and do
 |      something with it; i.e. send it up to the application or
 |      put it on the reassembly queue.
 */
static int In_Content_Process (Tcb, Msg, inp)
    TcbPtr Tcb;
    MsgPtr Msg;
    InPtr inp;
  {
    if (In_Ack_Process (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Window_Update (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    if (In_Data_Process (Tcb, Msg, inp) == I_STOP)
        return I_STOP;

    return I_NEXT;
  }

/* ------------------------------------------------------------------ */

/*  Input_Process
 |
 |  The input process takes a Tcb and a Msg; it first needs to ensure
 |  that the message passes the initial processing steps which consist
 |  mostly of validity checking. If this succeeds, then the content
 |  of the message is processed; this encompasses ack processing and
 |  actual data processing. Having finished content processing, we
 |  may need to do something.
 */
static void Input_Process (Tcb, Msg)
    TcbPtr Tcb;
    MsgPtr Msg;
  {
    Input_Info inp;

    inp.needoutput = FALSE;
    inp.acked = 0;

    if (In_Initial_Process (Tcb, Msg, &inp) == I_NEXT)
      {
        if (Msg->Flag_Ack == TRUE)
          {
            In_Content_Process (Tcb, Msg, &inp);

            if (inp.needoutput == TRUE || Tcb->Flag_Ack == TRUE)
              {
                Output_Process (Tcb, FALSE);
              }
          }
      }
  }

/* ------------------------------------------------------------------ */

```

2.3.4.8.8. Output

```

/* ------------------------------------------------------------------ */
/* $Id: tcp_output.c,v 1.3 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_output.c,v $
 * Revision 1.3  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.2  1995/10/10  08:15:17  mgream
 * cosmetic changes
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
/* Required Externals:
    OutQueue_EnQueue
    _BONeS_Get_Send_Buffer_Sz
    MsgCreate
 */

/* -------------------------------------------------------------------- */
/*  - - - OUTPUT PROCESSING - - -
 |
 |  Output processing involves two stages; the first is a check to
 |  determine whether there should be any output, and the second is
 |  concerned with actually carrying out the output.
 */
/* -------------------------------------------------------------------- */
/* -------------------------------------------------------------------- */

/*  EXIT_FLAGS
 |
 |  These flags are used as function returns to indicate appropriate
 |  action.
 */
#   define  O_STOP          0
#   define  O_SEND          1
#   define  O_NEXT          2

/* -------------------------------------------------------------------- */

/*  Output_Info_ST
 |
 |  This data structure is used to contain local information used
 |  during the Output phase; it was considered better to do it this
 |  way rather than pass a lot of variables through subroutines.
 */
typedef struct Output_Info_ST {
    boolean force;
    int idle;
    boolean sendalot;
    int off;
    long win;
    boolean ack_flag;
    long len;
  } Output_Info;
typedef Output_Info * OutPtr;

/* -------------------------------------------------------------------- */

/*  Out_Check_Forced
 |
 |  Here, we do some processing that occurs only if we are forcing
 |  an output; remember that the only condition for a forced output
 |  is during a window probe when we are persisting. So, what we do
 |  is ensure that we are sending _something_, even if it is only
 |  a size of one. However, the case may be that our window is not
 |  zero, therefore we can kill the persist timer.
 */
static int Out_Check_Forced (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    if (out->force == TRUE)
      {
        if (out->win == 0)
          {
            out->win = 1;
          }
        else
```

```
          {
            Tcb->Timer_Persist = 0;
            Tcb->t_rxtshift = 0;
          }
      }

    return O_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  Out_Compute_Size
 |
 |  Here, we figure out how much data we want to send. Firstly, we
 |  compute the initial size as the minimum of the send buffer and
 |  the available window; from that we subtract the amount that we
 |  have already send in this window. After which; we check for a
 |  negative length and do a check to see whether we are finished
 |  retransmitting. Finally, we truncate to maximum segment size
 |  that we are allowed to send, and make a note to the effect that
 |  we can come back here and send more.
 */
static int Out_Compute_Size (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    out->len = MIN (_BONeS_Get_Send_Buffer_Sz (Tcb), out->win) - out->off;

    if (out->len < 0)
      {
        out->len = 0;

        if (out->win == 0)
          {
            Tcb->Timer_Retransmit = 0;
            Tcb->snd_nxt = Tcb->snd_una;
          }
      }

    if (out->len > Tcb->t_maxseg)
      {
        out->len = Tcb->t_maxseg;
        out->sendalot = TRUE;
      }

    return O_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  Out_Silly_Window_Syndrome
 |
 |  Silly Window Syndrome Avoidance is carried out both by the sender
 |  and receiver; here we see the sender side of it. What occurs is
 |  that a set of conditions are checked to see whether sending a
 |  segment is OK. Note that this only occurs when we actually have
 |  data to send (i.e. not a window update or ack). The conditions
 |  that are checked for are:
 |  1. We are sending a maximum sized segment.
 |  2. We have been idle and we are depleting the output buffer.
 |  3. We are forcing output.
 |  4. We are sending more than half the maximum segment sent.
 |  5. We are retransmitting.
 */
static int Out_Silly_Window_Syndrome (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    if (out->len != 0)
      {
        if (out->len == Tcb->t_maxseg)
            return O_SEND;

        if (out->idle != 0 && out->len + out->off >=
                _BONeS_Get_Send_Buffer_Sz (Tcb))
            return O_SEND;

        if (out->force == TRUE)
            return O_SEND;

        if (out->len >= (Tcb->max_sndwnd / 2))
```

```
            return O_SEND;

        if (SEQ_LT (Tcb->snd_nxt, Tcb->snd_max))
            return O_SEND;
      }

    return O_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  Out_Window_Update
 |
 |  Check to see whether we are sending a pure window update. What we
 |  do is see whether the advertised window has changed by at least
 |  two maximum segments. Note that in this simulation, some of this
 |  code will never be executed; i.e. it should _always_ escape with
 |  O_SEND. The reason it has been left in is to preserve the logical
 |  structure and allow for a future modification.
 */
static int Out_Window_Update (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    if (out->win > 0)
      {
        long adv;

        adv = MIN (out->win, (long)(TCP_MAXWIN << Tcb->rcv_scale));
        adv = adv - (Tcb->rcv_adv - Tcb->rcv_nxt);

        if (adv >= (long)(2 * Tcb->t_maxseg))
            return O_SEND;
      }

    return O_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  Out_Flags
 |
 |  We may be explicitly sending an Acknowledgement, so make sure we
 |  go and send if this is the case.
 */
static int Out_Flags (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    if (out->ack_flag == TRUE)
      {
        return O_SEND;
      }

    return O_NEXT;
  }

/* -------------------------------------------------------------------- */

/*  Out_Persist_Check
 |
 |  Here, we look at whether or not we are in the persist state; which
 |  occurs the buffer size is greater than zero, and we have failed
 |  all the previous output conditions. So, the persist timer is set
 |  up here then.
 */
static int Out_Persist_Check (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    if (_BONeS_Get_Send_Buffer_Sz (Tcb) > 0)
      {
        if (Tcb->Timer_Retransmit == 0 && Tcb->Timer_Persist == 0)
          {
            Tcb->t_rxtshift = 0;
            Timer_Persist_Setup (Tcb);
          }
      }

    return O_NEXT;
  }
```

```

/* ------------------------------------------------------------------ */

/*  Out_Check_If_Output_Needed
 |
 |  This is the first half of tcp output processing, where we actually
 |  try to determine whether or not we should send something, and if
 |  so then establish the basic parameters (i.e. amount to send and so
 |  forth). Each subroutine indicates whether or not it has exited
 |  because it wants to SEND, or STOP, or CONTINUE (NEXT). We execute
 |  each check in succession. The checks are as follows:
 |  1. Forced Output -- there are some special conditions that occur
 |      if we are _forcing_ output, so we do these.
 |  2. Compute Size -- determine how much data we have to send, within
 |      the constraints of window, buffer and other sizes.
 |  3. Silly Window Syndrome -- check out the silly window syndrome
 |      conditions; these may or may not inhibit transmission.
 |  4. Window Update -- we may be sending a window update, so do it
 |      in here if that is the case.
 |  5. Flags Check -- certain specific flags; i.e. "ack" may require
 |      us to send.
 |  6. Persist Check -- finally, we may need to persist to probe for
 |      a window change.
 */
static int Out_Check_If_Output_Needed (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    if (Out_Check_Forced (Tcb, out) == O_SEND)
        return O_SEND;

    if (Out_Compute_Size (Tcb, out) == O_SEND)
        return O_SEND;

    out->win = TCP_MAXWIN << TCP_MAX_WINSHIFT;

    if (Out_Silly_Window_Syndrome (Tcb, out) == O_SEND)
        return O_SEND;

    if (Out_Window_Update (Tcb, out) == O_SEND)
        return O_SEND;

    if (Out_Flags (Tcb, out) == O_SEND)
        return O_SEND;

    if (Out_Persist_Check (Tcb, out) == O_SEND)
        return O_SEND;

    return O_STOP;
  }

/* ------------------------------------------------------------------ */

/*  Out_Construct_Output_Msg
 |
 |  Construct the output TCP message by filling in all the appropriate
 |  fields; this includes length, sequence number, flags, windows and
 |  timestamps.
 */
static int Out_Construct_Output_Msg (Tcb, out, Msg)
    TcbPtr Tcb;
    OutPtr out;
    MsgPtr Msg;
  {
    /* Length */
    Msg->len = (out->len > 0) ? out->len : 0;

    /* Sequence Number */
    if (out->len > 0 || Tcb->Timer_Persist != 0)
        Msg->seq = Tcb->snd_nxt;
    else
        Msg->seq = Tcb->snd_max;
    Msg->ack = Tcb->rcv_nxt;

    /* Flags */
    Msg->Flag_Ack = out->ack_flag;

    /* Window */
    if (out->win < Tcb->t_maxseg)
        out->win = 0;
    if (out->win > (long)(TCP_MAXWIN << Tcb->rcv_scale))
```

```
        out->win = (long)(TCP_MAXWIN << Tcb->rcv_scale);
    Msg->win = out->win >> Tcb->rcv_scale;

    /* Timestamps */
    Msg->Flag_Timestamp = Tcb->Flag_Timestamp;
    Msg->t_now = Tcb->tcp_now;
    Msg->t_recent = Tcb->ts_recent;

    return O_SEND;
  }

/* ------------------------------------------------------------------- */

/*  Out_Send_Msg
 |
 |  The sending is done here, all we do is queue the message using a
 |  defined primitive. Messages are then dequeued just as we go back
 |  to the BONeS environment.
 */
static int Out_Send_Msg (Tcb, out, Msg)
    TcbPtr Tcb;
    OutPtr out;
    MsgPtr Msg;
  {
    OutQueue_EnQueue (Msg);

    return O_SEND;
  }

/* ------------------------------------------------------------------- */

/*  Out_Update_Sequence_Numbers
 |
 |  Having just sent the message, we must update the various sequence
 |  numbers such as the maximum sequence number sent, and that sent
 |  but not acknowledged. What we do here is first check to see whether
 |  we are outputing because we are not forced or retransmitting, and
 |  then first update the maximum and next sequence numbers, setting
 |  up a rtt timer (i.e. the rtt timer only occurs if we are sending
 |  new data, not retransmitting). We make sure we setup for another
 |  retransmit too, if we are retransmitting.
 */
static int Out_Update_Sequence_Numbers (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    if (out->force == FALSE || Tcb->Timer_Persist == 0)
      {
        if (SEQ_GT (Tcb->snd_nxt + out->len, Tcb->snd_max))
          {
            Tcb->snd_max = Tcb->snd_nxt + out->len;

            if (Tcb->t_rtt == 0)
              {
                Tcb->t_rtt = 1;
                Tcb->t_rtseq = Tcb->snd_nxt;
              }
          }

        Tcb->snd_nxt = Tcb->snd_nxt + out->len;

        if (Tcb->Timer_Retransmit == 0 && Tcb->snd_nxt != Tcb->snd_una)
          {
            Tcb->Timer_Retransmit = Tcb->t_rxtcur;

            if (Tcb->Timer_Persist != 0)
              {
                Tcb->Timer_Persist = 0;
                Tcb->t_rxtshift = 0;
              }
          }
      }
    else
      {
        if (SEQ_GT (Tcb->snd_nxt + out->len, Tcb->snd_max))
          {
            Tcb->snd_max = Tcb->snd_nxt + out->len;
          }
      }

    return O_SEND;
  }

/* ------------------------------------------------------------------- */

/*  Out_Send_Output
 |
 |  The second half of output processing is to actuall construct and
 |  send a message, then to send it and update state variables in the
 |  TCB. This is done in three steps, first the message is constructed,
 |  then it is sent, and finally the various sequence numbers and the
 |  such like are updated.
 */
static int Out_Send_Output (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    MsgPtr Msg = MsgCreate ();
    Out_Construct_Output_Msg (Tcb, out, Msg);
    Out_Send_Msg (Tcb, out, Msg);
    Out_Update_Sequence_Numbers (Tcb, out);

    return O_SEND;
  }

/* ------------------------------------------------------------------- */

/*  Out_First_Init
 |
 |  Output processing will iterate if there are a number of segments to
 |  send. So, at the start we must do some very first initialising to
 |  set up a few things. We set up the forced output flag, the idle
 |  flag, and if we have been idle then we reset the congestion window.
 */
static void Out_First_Init (Tcb, out, Force)
    TcbPtr Tcb;
    OutPtr out;
    boolean Force;
  {
    out->force = Force;

    out->idle = (Tcb->snd_max == Tcb->snd_una);

    if (out->idle != 0 && Tcb->t_idle >= Tcb->t_rxtcur)
      {
        Tcb->snd_cwnd = Tcb->t_maxseg;
      }
  }

/* ------------------------------------------------------------------- */

/*  Out_Loop_Init
 |
 |  Initialise per segment looping; i.e reset the iterator flag, and
 |  set up our window offset and window size and our ack flag.
 */
static void Out_Loop_Init (Tcb, out)
    TcbPtr Tcb;
    OutPtr out;
  {
    out->sendalot = FALSE;
    out->off = Tcb->snd_nxt - Tcb->snd_una;
    out->win = MIN (Tcb->snd_wnd, Tcb->snd_cwnd);
    out->ack_flag = Tcb->Flag_Ack;
  }

/* ------------------------------------------------------------------- */

/*  Output_Process
 |
 |  The complete output processing stage; what we do here is use a
 |  local structure to maintain some common parameters and then
 |  initialise these with global values for all the segments we will
 |  output. Following this, we continue to look whilst the loop flag
 |  is set, and on each loop we will output a message. The looping
 |  itself consists of two stages, the first of which is checking
 |  to see if there is any output, and then if there is, actually
 |  constructing and sending the otuput.
 */
static void Output_Process (Tcb, Force)
    TcbPtr Tcb;
    boolean Force;
  {
    Output_Info OInfo;

    Out_First_Init (Tcb, &OInfo, Force);

    do
      {
        Out_Loop_Init (Tcb, &OInfo);

        if (Out_Check_If_Output_Needed (Tcb, &OInfo) == O_STOP)
            break;

        Out_Send_Output (Tcb, &OInfo);
      }
    while (OInfo.sendalot == TRUE);
  }

/* --------------------------------------------------------------------- */

```

2.3.4.8.9. Quench

```

/* --------------------------------------------------------------------- */
/* $Id: tcp_quench.c,v 1.1 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_quench.c,v $
 * Revision 1.1  1995/12/21  11:08:30  mgream
 * Initial revision
 *
 */
/* --------------------------------------------------------------------- */

/* --------------------------------------------------------------------- */
/* Required Externals:
    OutQueue_EnQueue
    _BONeS_Get_Send_Buffer_Sz
    MsgCreate
 */

/* --------------------------------------------------------------------- */
/*  - - - QUENCH PROCESSING - - -
 |
 |  Very simple .. scale back the congestion window.
 */
/* --------------------------------------------------------------------- */
/* --------------------------------------------------------------------- */

/*  Quench_Process
 |
 |  Scale it back ...
 */
static void Quench_Process (Tcb)
    TcbPtr Tcb;
  {
    Tcb->snd_cwnd = Tcb->t_maxseg;
  }

/* --------------------------------------------------------------------- */

```

2.3.4.8.10. Timer

```

/* --------------------------------------------------------------------- */
/* $Id: tcp_timers.c,v 1.3 1995/12/21 11:08:30 mgream Exp $
 * $Log: tcp_timers.c,v $
 * Revision 1.3  1995/12/21  11:08:30  mgream
 * integration fixes -- namely small bug fixes and name mismatches
 *
 * Revision 1.2  1995/10/10  08:15:17  mgream
 * cosmetic changes
 *
 * Revision 1.1  1995/10/10  08:07:07  mgream
 * Initial revision
 *
 */
/* -------------------------------------------------------------------- */

/* -------------------------------------------------------------------- */
/* Required Externals:
 */

/* -------------------------------------------------------------------- */
/*  - - - TIMER PROCESSING - - -
 |
 |  We do the timer processing in here; our input is a single kick
 |  every 100ms which maps out to 200ms and 500ms kicks for the fast
 |  and slow timers respectively. The fast timer kicks delayed acks,
 |  and the slow timer kicks retransmit and persist processing. Any
 |  of these three may result in output messages being generated. In
 |  addition, there are some minor housekeeping functions that are
 |  performed (the slow timer counts rtt's and a monotonic virtual
 |  clock).
 */
/* -------------------------------------------------------------------- */
/* -------------------------------------------------------------------- */

/*  Timer_Process
 |
 |  Kick in here on 100ms timer expiries from BONeS which we thump
 |  down into 200ms or 500ms expiries that correspond with TCP's
 |  fast and slow timer. These timer handlers are then called if
 |  appropriate.
 */
static void Timer_Process (Tcb)
    TcbPtr Tcb;
  {
    Tcb->_timer_ticks = (Tcb->_timer_ticks + 1) % 10;

    if ((Tcb->_timer_ticks % 2) == 0)
      {
        Timer_Fast_Process (Tcb);
      }

    if ((Tcb->_timer_ticks % 5) == 0)
      {
        Timer_Slow_Process (Tcb);
      }
  }

/* -------------------------------------------------------------------- */

/*  Timer_Fast_Process
 |
 |  The fast timer is used to schedule delayed acks; so we check to
 |  see whether there is a delayed ack pending, and if so then go and
 |  pump it out via the output processing stage.
 */
static void Timer_Fast_Process (Tcb)
    TcbPtr Tcb;
  {
    if (Tcb->Flag_DelayedAck == TRUE)
      {
        Tcb->Flag_Ack = TRUE;
        Tcb->Flag_DelayedAck = FALSE;

        Output_Process (Tcb, FALSE);
      }
  }

/* -------------------------------------------------------------------- */

/*  Timer_Slow_Process
 |
 |  The slow timer is used to schedule retransmits and persists, so
 |  we check to see whether either of these timers have expired and
 |  if so, then go off and handle them. Also, we ensure that we update
 |  our idle counter (which is reset in input processing) and the
 |  round trip time if we are timing a segment. Also increase tcp_now
 |  which acts as a time counter for all TCP processing.
 */
static void Timer_Slow_Process (Tcb)
```

```
    TcbPtr Tcb;
  {
    if (Tcb->Timer_Retransmit > 0)
      {
        if (--Tcb->Timer_Retransmit == 0)
          {
            Timer_Retransmit_Process (Tcb);
          }
      }

    if (Tcb->Timer_Persist > 0)
      {
        if (--Tcb->Timer_Persist == 0)
          {
            Timer_Persist_Process (Tcb);
          }
      }

    Tcb->t_idle++;

    if (Tcb->t_rtt > 0)
      {
        Tcb->t_rtt++;
      }

    Tcb->tcp_now++;
  }

/* --------------------------------------------------------------------- */

/*  Get_Backoff_Value
 |
 |  Compute a backoff value according to a specific shift; this is a
 |  base-2 exponential backoff, constrained at 6 bits. This is
 |  independant of the TCB.
 */
static int Get_Backoff_Value (shift)
    int shift;
  {
    return MIN (1 << shift, 1 << 6);
  }

/* --------------------------------------------------------------------- */

/*  Get_Retransmit_Value
 |
 |  Compute the retransmit value using the smoothed round trip time
 |  and the round trip variance.
 */
static int Get_Retransmit_Value (Tcb)
    TcbPtr Tcb;
  {
    return (Tcb->t_srtt >> TCP_RTT_SHIFT) + Tcb->t_rttvar;
  }

/* --------------------------------------------------------------------- */

/*  Confine_Range
 |
 |  Confine a value to be between a minimum and maximum. This is
 |  indepedant of the TCB.
 */
static int Confine_Range (Value, Min, Max)
    int Value;
    int Min;
    int Max;
  {
    if (Value < Min)
      {
        return Min;
      }
    else if (Value > Max)
      {
        return Max;
      }
    else
      {
        return Value;
      }
  }

```

```
/* -------------------------------------------------------------------- */

/*  Rxt_Update_Backoff
 |
 |  In processing the retransmit timer, we need to update the
 |  retransmit backoff value.
 */
static void Rxt_Update_Backoff (Tcb)
    TcbPtr Tcb;
  {
    if (++Tcb->t_rxtshift > TCP_MAXRXTSHIFT)
      {
        Tcb->t_rxtshift = TCP_MAXRXTSHIFT;
      }
  }

/* -------------------------------------------------------------------- */

/*  Rxt_Setup_Next_Timer
 |
 |  We need to schedule another retransmit timer by computing the time
 |  according to our round trip time. We also reset the send sequence
 |  to be the start of our unacknowledged data, and reset the round
 |  trip time because it is not valid anymore.
 */
static void Rxt_Setup_Next_Timer (Tcb)
    TcbPtr Tcb;
  {
    int rxtval;

    rxtval = Get_Retransmit_Value (Tcb) * Get_Backoff_Value (Tcb->t_rxtshift);

    Tcb->t_rxtcur = Confine_Range (rxtval, Tcb->t_rttmin, TCPTV_REXMTMAX);
    Tcb->Timer_Retransmit = Tcb->t_rxtcur;

    if (Tcb->t_rxtshift > (TCP_MAXRXTSHIFT / 4))
      {
        Tcb->t_rttvar += (Tcb->t_srtt >> TCP_RTT_SHIFT);
        Tcb->t_srtt = 0;
      }

    Tcb->snd_nxt = Tcb->snd_una;
    Tcb->t_rtt = 0;
  }

/* -------------------------------------------------------------------- */

/*  Rxt_Update_Congestion_Information
 |
 |  Scale down the congestion window, because we have lost data that
 |  was in the pipe. Also, reset duplicate acks count and so on.
 */
static void Rxt_Update_Congestion_Information (Tcb)
    TcbPtr Tcb;
  {
    u_int win;

    win = MIN (Tcb->snd_wnd, Tcb->snd_cwnd) / 2 / Tcb->t_maxseg;
    if (win < 2)
        win = 2;

    Tcb->snd_cwnd = Tcb->t_maxseg;
    Tcb->snd_ssthresh = win * Tcb->t_maxseg;
    Tcb->t_dupacks = 0;
  }

/* -------------------------------------------------------------------- */

/*  Timer_Retransmit_Process
 |
 |  When a retransmit timer expires, then first update our backoff
 |  value, schedule a another timer event and fix up the congestion
 |  state. After which we call output processing to start pumping
 |  data back into the pipe.
 */
static void Timer_Retransmit_Process (Tcb)
    TcbPtr Tcb;
  {
    Rxt_Update_Backoff (Tcb);
    Rxt_Setup_Next_Timer (Tcb);
    Rxt_Update_Congestion_Information (Tcb);

    Output_Process (Tcb, FALSE);
  }

/* ------------------------------------------------------------------ */

/*  Timer_Persist_Setup
 |
 |  Setup the persist timer; we do this by looking at the round trip
 |  time mean and its variance, and our computed backoff value. The
 |  persist timer is then scheduled and the backoff increases for the
 |  next persist; should it come around.
 */
static void Timer_Persist_Setup (Tcb)
    TcbPtr Tcb;
  {
    int perval;

    perval = (((Tcb->t_srtt >> 2) + Tcb->t_rttvar) >> 1) *
             Get_Backoff_Value (Tcb->t_rxtshift);
    Tcb->Timer_Persist = Confine_Range (perval, TCPTV_PERSMIN, TCPTV_PERSMAX);

    if (Tcb->t_rxtshift < TCP_MAXRXTSHIFT)
      {
        Tcb->t_rxtshift++;
      }
  }

/* ------------------------------------------------------------------ */

/*  Timer_Persist_Process
 |
 |  Process the persist timer, all we do here is setup another persist
 |  timer and kick output processing with an indication that we want to
 |  force output.
 */
static void Timer_Persist_Process (Tcb)
    TcbPtr Tcb;
  {
    Timer_Persist_Setup (Tcb);

    Output_Process (Tcb, TRUE);
  }

/* ------------------------------------------------------------------ */

```

2.4. Network-Adaption Layer

2.4.1. Data Structures

2.4.1.1. IE Network-Adaption Primitive

This Data Structure has no content.

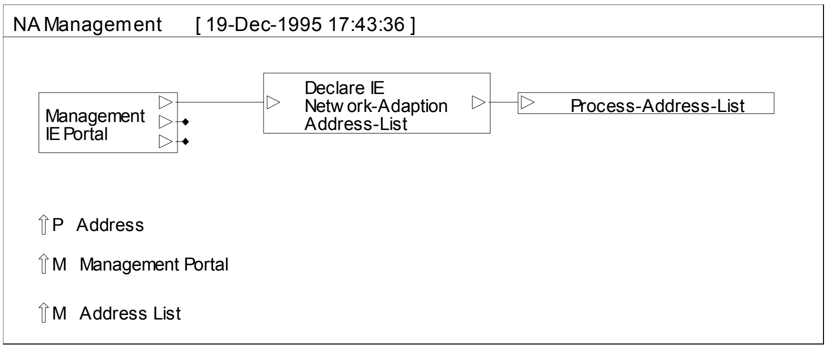
2.4.1.2. IE Network-Adaption Address List

Name	*Type*	*Subrange*	*Default Value*
Address List	INT-VECTOR		

2.4.2. Main Modules

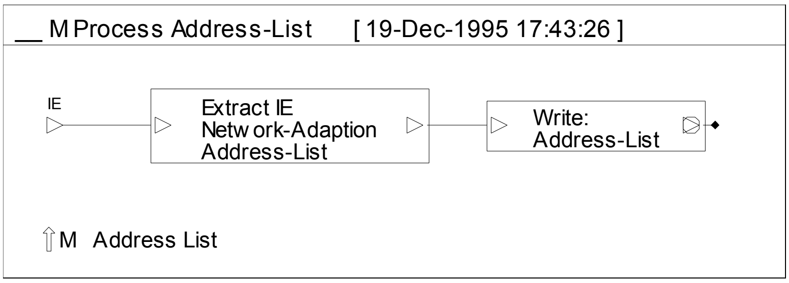
2.4.2.1. Management

This Module implements "DFD 2: Management Processor". This also incorporates "PSPEC 2.1: Validate Mgmt Message and Extract IE".



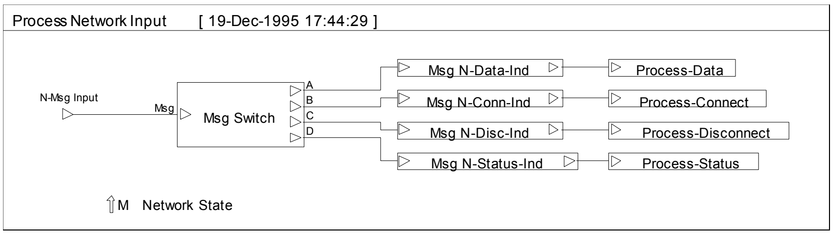
2.4.2.2. Management -- Process Address List

This Module implements "PSPEC 2.2: Process Address List IE".


__ M Process Address-List [19-Dec-1995 17:43:26]
IE
Extract IE Netw ork-Adaption Address-List
Write: Address-List
M Address List

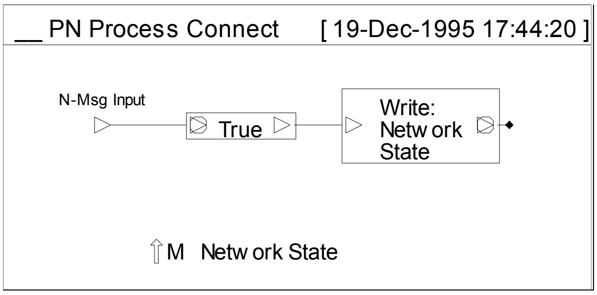
2.4.2.3. Process Network Input

This Module implements "DFD 1: Process Network Message". This includes "PSPEC 1.1: Classify Network Message".



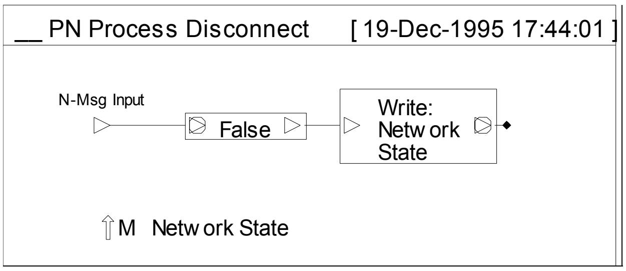
2.4.2.4. Process Network Input -- Process Connect

This Module implements “PSPEC 1.2: Process Connect Message”.



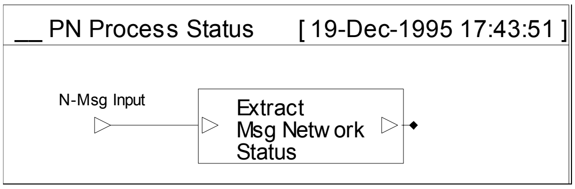
2.4.2.5. Process Network Input -- Process Disconnect

This Module implements “PSPEC 1.3: Process Disconnect Message”.



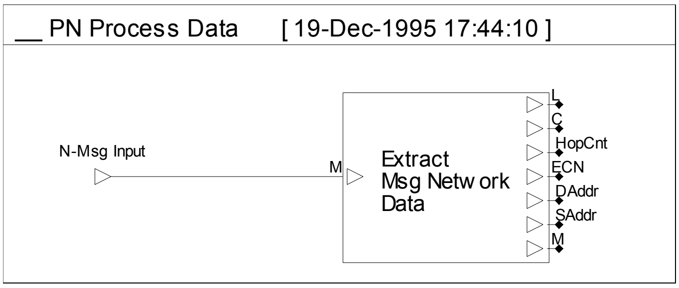
2.4.2.6. Process Network Input -- Process Status

This Module implements “PSPEC 1.4: Process Status Message”.


__ PN Process Status [19-Dec-1995 17:43:51]
N-Msg Input
Extract Msg Netw ork Status

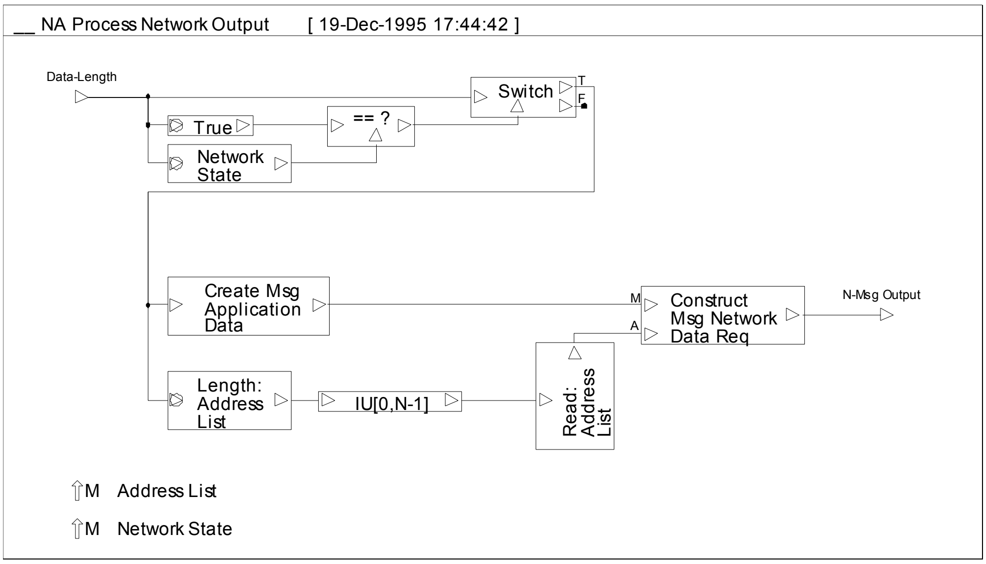
2.4.2.7. Process Network Input -- Process Data

This Module implements “PSPEC 1.5: Process Data Message”.


__ PN Process Data [19-Dec-1995 17:44:10]
N-Msg Input
M
Extract Msg Netw ork Data
L
C
HopCnt
ECN
DAddr
SAddr
M

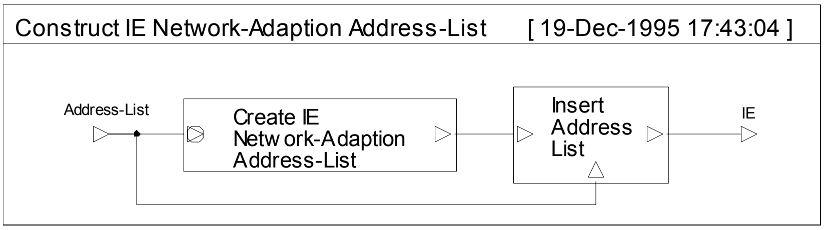
2.4.2.8. Process Network Output

This Module implements “PSPEC 3: Construct Outgoing Message”.

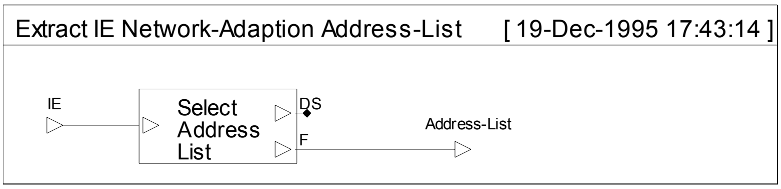


2.4.3. Support Modules

2.4.3.1. Construct IE Network-Adaption Address List


Construct IE Network-Adaption Address-List [19-Dec-1995 17:43:04]
Address-List
Create IE Netw ork-Adaption Address-List
Insert Address List
IE

2.4.3.2. Extract IE Network-Adaption Address List


Extract IE Network-Adaption Address-List [19-Dec-1995 17:43:14]
IE
Select Address List
DS
F
Address-List

2.5. Transport-Adaption Layer

2.5.1. Data Structures

2.5.1.1. IE Transport-Adapation Primitive

This Data Structure has no content.

2.5.1.2. IE Transport-Adaption Connect

Name	*Type*	*Subrange*	*Default Value*
Destination Address	INTEGER	[0,512)	0

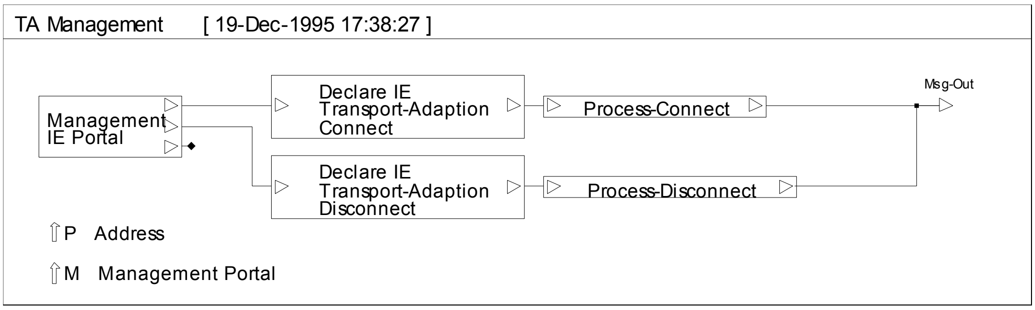
2.5.1.3. IE Transport-Adaption Disconnect

This Data Structure has no content.

2.5.2. Main Modules

2.5.2.1. Management

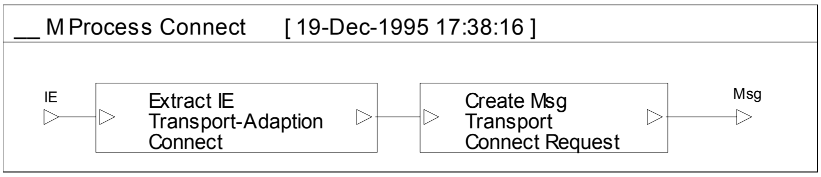
This Module implements “DFD 2: Management Processor”. This also includes “PSPEC 2.1: Validate Mgmt Message and Extract IE”.



TA Management [19-Dec-1995 17:38:27]
Management IE Portal
Declare IE Transport-Adaption Connect
Declare IE Transport-Adaption Disconnect
Process-Connect
Process-Disconnect
Msg-Out
P Address
M Management Portal

2.5.2.2. Management -- Process Connect

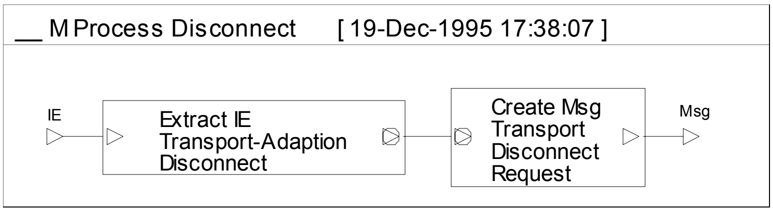
This Module implements “PSPEC 2.2: Process Connect IE”.



__ M Process Connect [19-Dec-1995 17:38:16]
IE
Extract IE Transport-Adaption Connect
Create Msg Transport Connect Request
Msg

2.5.2.3. Management -- Process Disconnect

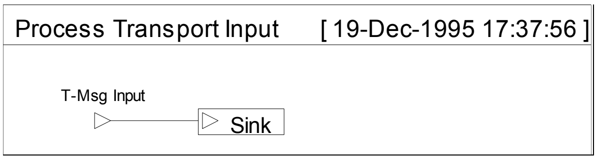
This Module implements “PSPEC 2.3: Process Disconnect IE”.



__ M Process Disconnect [19-Dec-1995 17:38:07]
IE
Extract IE Transport-Adaption Disconnect
Create Msg Transport Disconnect Request
Msg

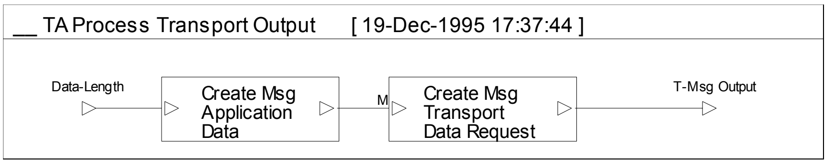
2.5.2.4. Process Transport Input

This Module implements “DFD 1: Process Transport Message”. This includes “PSPEC 1.1: Classify Transport Message”, “PSPEC 1.2: Process Connect Message”, “PSPEC 1.3: Process Disconnect Message” and “PSPEC 1.4: Process Data Message”.



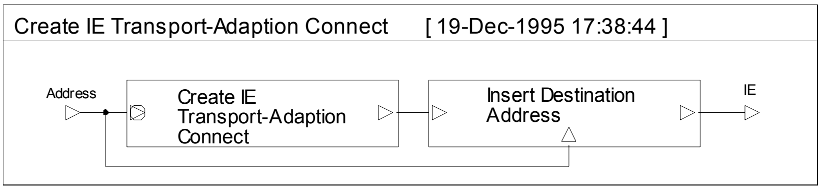
2.5.2.5. Process Transport Output

This Module implements “PSPEC 3: Construct Outgoing Message”.

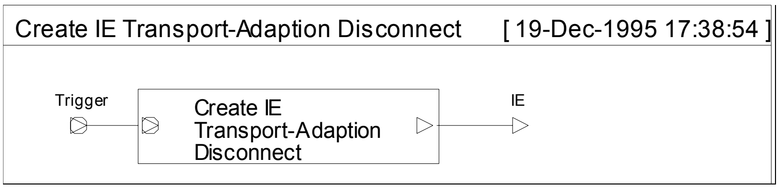

__ TA Process Transport Output [19-Dec-1995 17:37:44]
Data-Length
Create Msg Application Data
M
Create Msg Transport Data Request
T-Msg Output

2.5.3. Support Modules

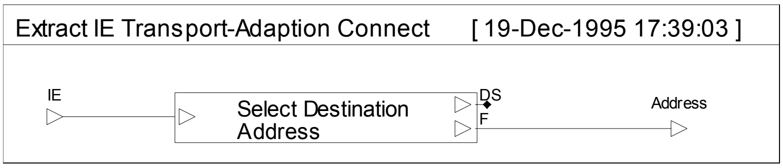
2.5.3.1. Create IE Transport-Adapation Connect


Create IE Transport-Adaption Connect [19-Dec-1995 17:38:44]
Address
Create IE Transport-Adaption Connect
Insert Destination Address
IE

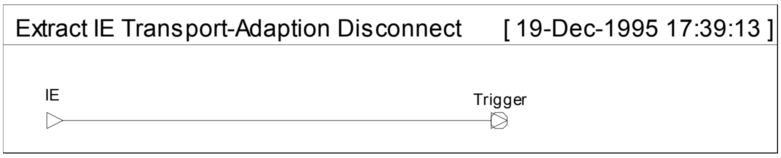
2.5.3.2. Create IE Transport-Adaption Disconnect


Create IE Transport-Adaption Disconnect [19-Dec-1995 17:38:54]
Trigger
Create IE Transport-Adaption Disconnect
IE

2.5.3.3. Extract IE Transport-Adaption Connect


Extract IE Transport-Adaption Connect [19-Dec-1995 17:39:03]
IE
Select Destination Address
DS
F
Address

2.5.3.4. Extract IE Transport-Adaption Disconnect


Extract IE Transport-Adaption Disconnect [19-Dec-1995 17:39:13]
IE
Trigger

2.6. Routing-Module

2.6.1. Data Structures

2.6.1.1. IE Routing-Module Primitive

This Data Structure has no content.

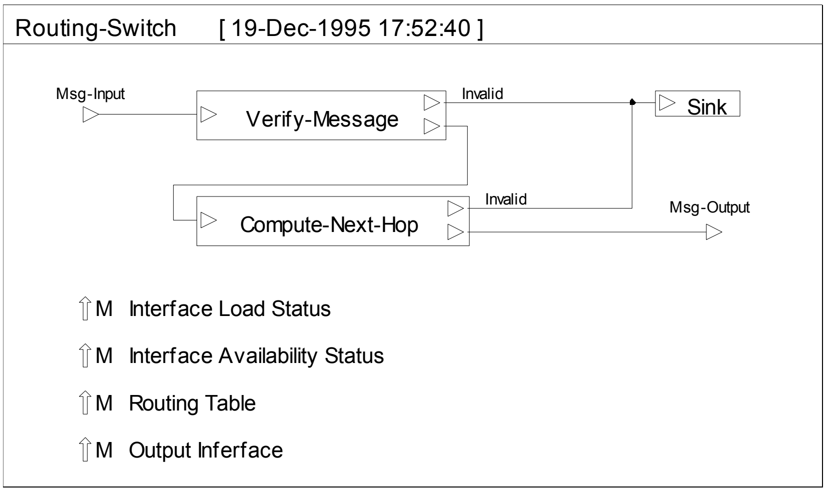
2.6.1.2. IE Routing-Module Route-Entry

Name	*Type*	*Subrange*	*Default Value*
End-System Address	INTEGER	[0,512)	0
Network Interface	INTEGER	[0,512)	0
Cost	INTEGER	(-Inf,+Inf)	0

2.6.2. Main Modules

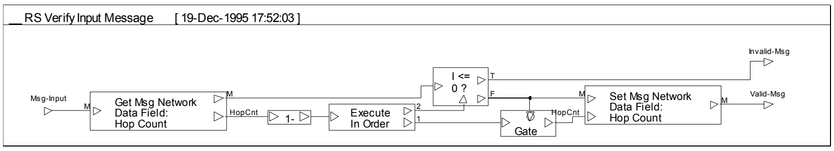
2.6.2.1. Routing Switch

This Module implements "DFD 1: Routing Module". This also incorporates "PSPEC 1.2: Drop Invalid Message".



2.6.2.2. Routing Switch -- Verify Input Message

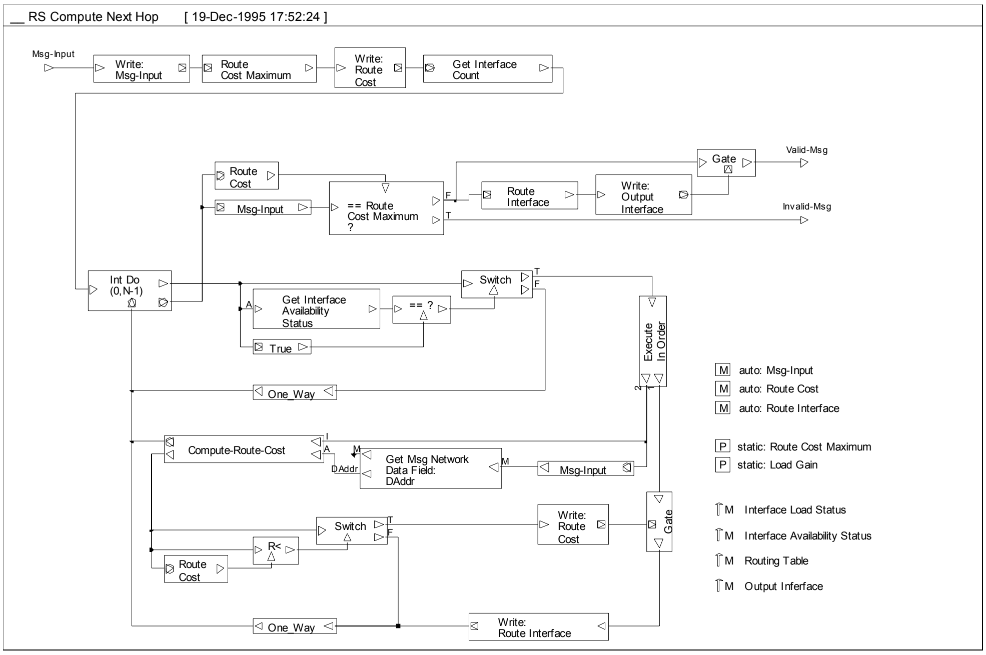
This Module implements "PSPEC 1.1: Verify and Update Incoming Message".



RS Verify Input Message [19-Dec-1995 17:52:03]
Msg-Input
M
Get Msg Network Data Field: Hop Count
M
HopCnt
1-
Execute In Order
2
1
I <= 0 ?
T
F
Gate
HopCnt
M
Set Msg Network Data Field: Hop Count
M
Invalid-Msg
Valid-Msg

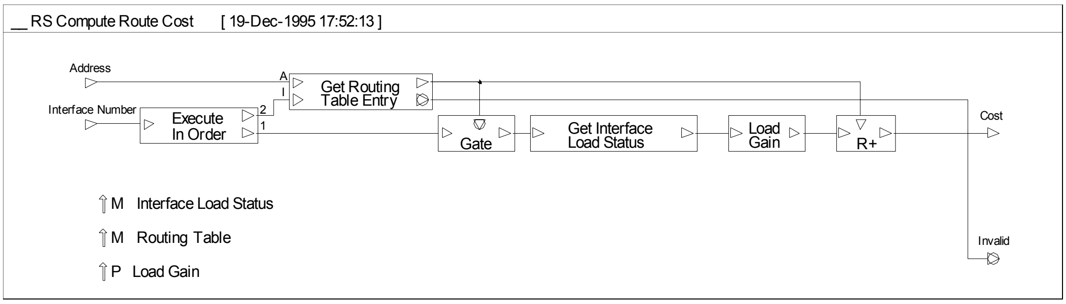
2.6.2.3. Routing Switch -- Compute Next Hop

This Module implements "PSPEC 1.3: Compute Next Hop". This BONeS implementation is a potential target for 'C' implementation, as the iterative mechanism is repeated for every Message that the Routing Module processes.



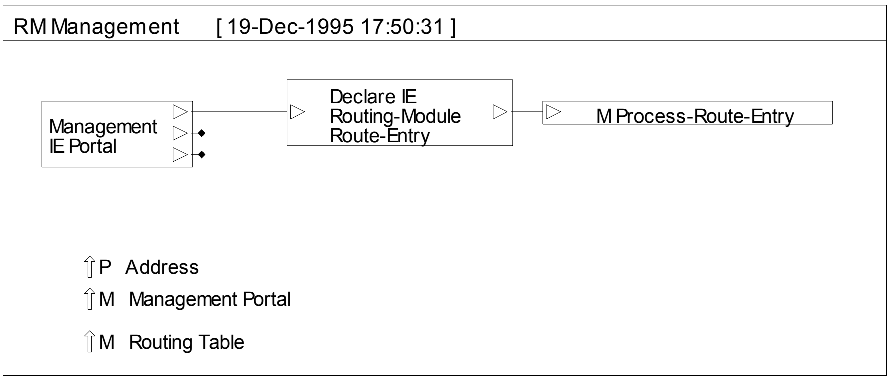
2.6.2.4. Routing Switch -- Compute Next Hop -- Compute Route Cost

This Module implements “FUNCTION 1.3.1: ComputeCost”. The “Load Gain” is a parameter as specified in the algorithm given in the design.



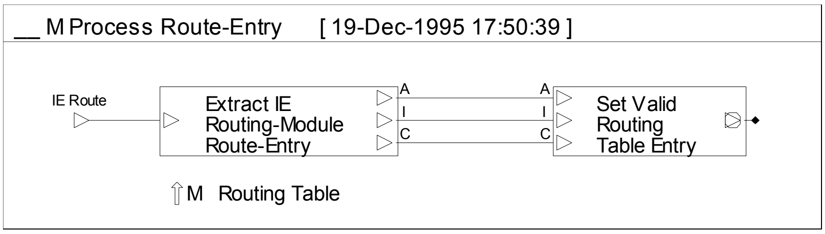
2.6.2.5. Management

This Module implements “DFD 2: Management Processor”. This also incorporates “PSPEC 2.1: Validate Mgmt Message and Extract IE”.



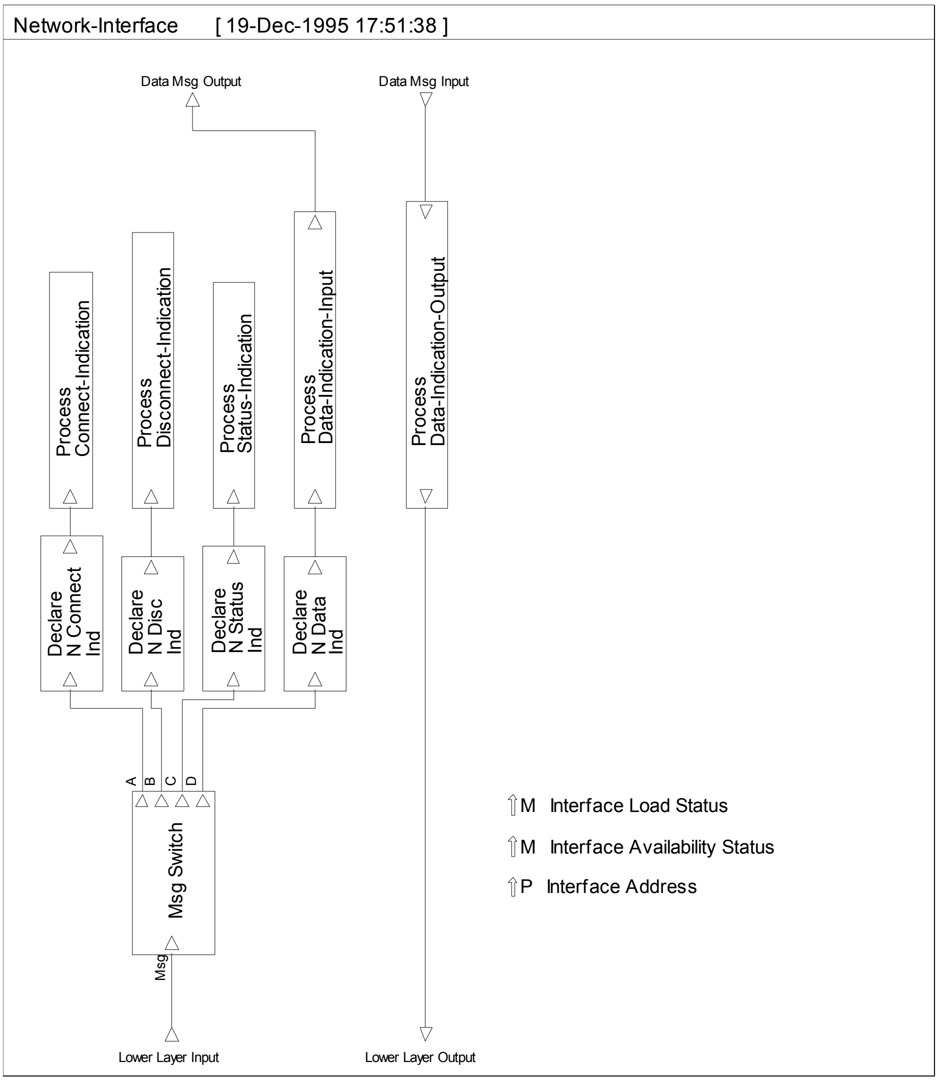
2.6.2.6. Management -- Process Route Entry

This Module implements “PSPEC 2.2: Process Routing Entry IE”.



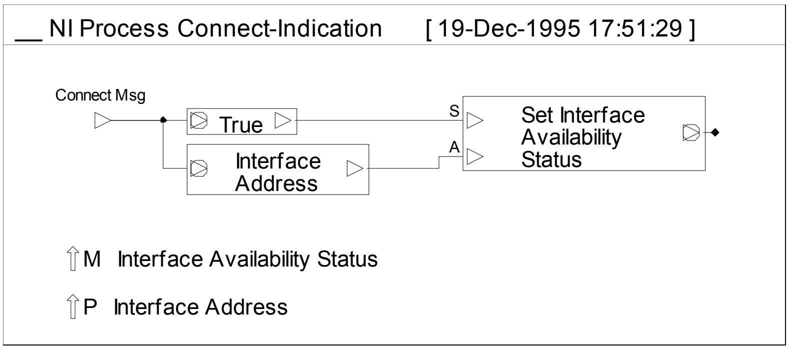
2.6.2.7. Network Interface

This Module implements “DFD 4: Network Layer Interface”.



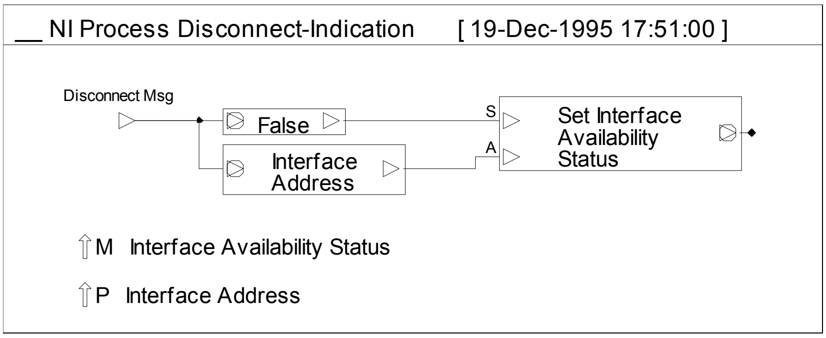
2.6.2.8. Network Interface -- Process Connect Indication

This Module implements “PSPEC 4.2: Process Connect Message”.


__ NI Process Connect-Indication [19-Dec-1995 17:51:29]
Connect Msg
True
Interface Address
S
A
Set Interface Availability Status
M Interface Availability Status
P Interface Address

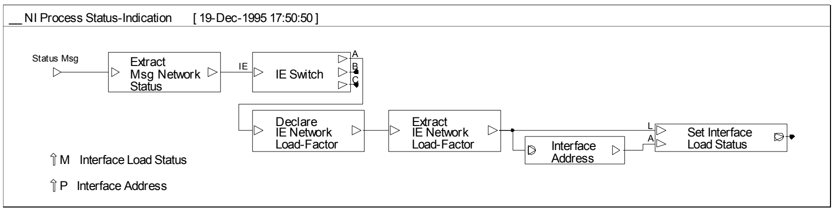
2.6.2.9. Network Interface -- Process Disconnect Indication

This Module implements “PSPEC 4.3: Process Disconnect Message”.


__ NI Process Disconnect-Indication [19-Dec-1995 17:51:00]
Disconnect Msg
False
Interface Address
S
A
Set Interface Availability Status
M Interface Availability Status
P Interface Address

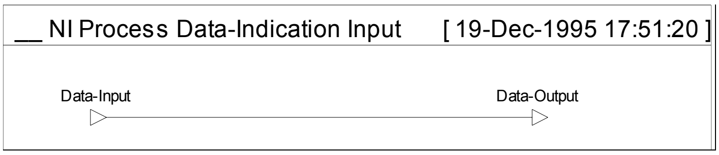
2.6.2.10. Network Interface -- Process Status Indication

This Module implements “PSPEC 4.4: Process Status Message”.


__ NI Process Status-Indication [19-Dec-1995 17:50:50]
Status Msg
Extract Msg Network Status
IE
IE Switch
A
B
C
Declare IE Network Load-Factor
Extract IE Network Load-Factor
Interface Address
L
A
Set Interface Load Status
M Interface Load Status
P Interface Address

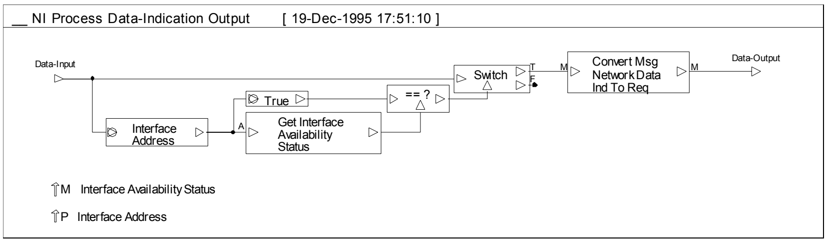
2.6.2.11. Network Interface -- Process Data Indication Input

This Module implements “PSPEC 4.5: Process Data Message”.

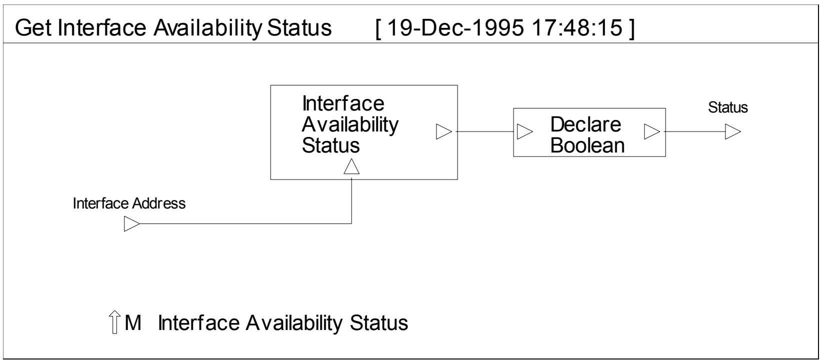

__ NI Process Data-Indication Input [19-Dec-1995 17:51:20]
Data-Input
Data-Output

2.6.2.12. Network Interface -- Process Data Indication Output

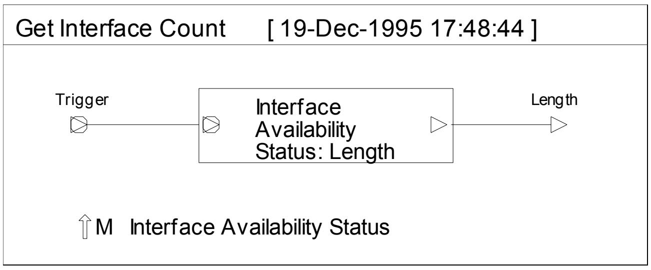
This Module implements “PSPEC 4.6: Process Outgoing Data Message”.


__ NI Process Data-Indication Output [19-Dec-1995 17:51:10]
Data-Input
Interface Address
A
True
Get Interface Availability Status
== ?
Switch
T
F
M
Convert Msg Network Data Ind To Req
M
Data-Output
M Interface Availability Status
P Interface Address

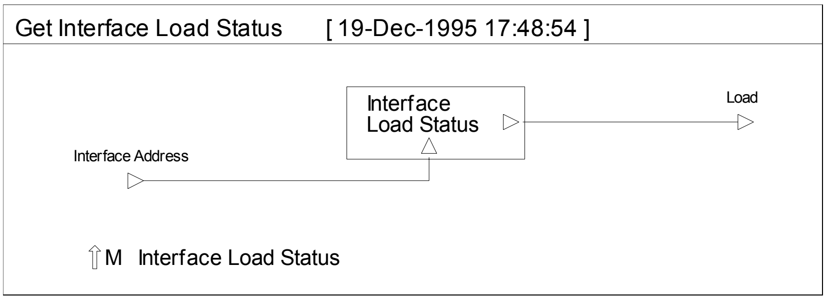
2.6.2.13. Get Interface Availability Status


Get Interface Availability Status [19-Dec-1995 17:48:15]
Interface Availability Status
Declare Boolean
Status
Interface Address
⇧M Interface Availability Status

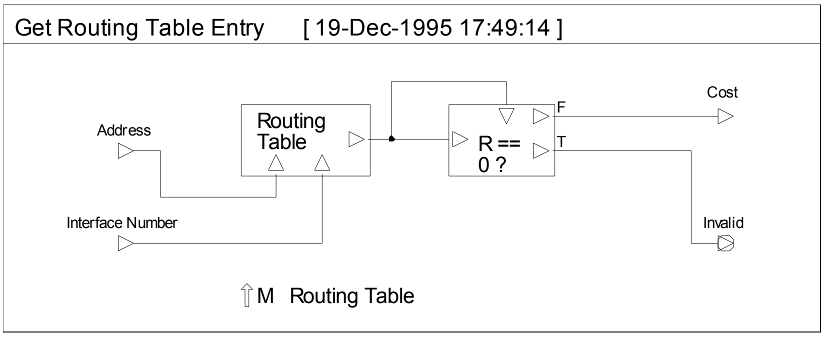
2.6.2.14. Get Interface Count


Get Interface Count [19-Dec-1995 17:48:44]
Trigger
Interface Availability Status: Length
Length
⇧M Interface Availability Status

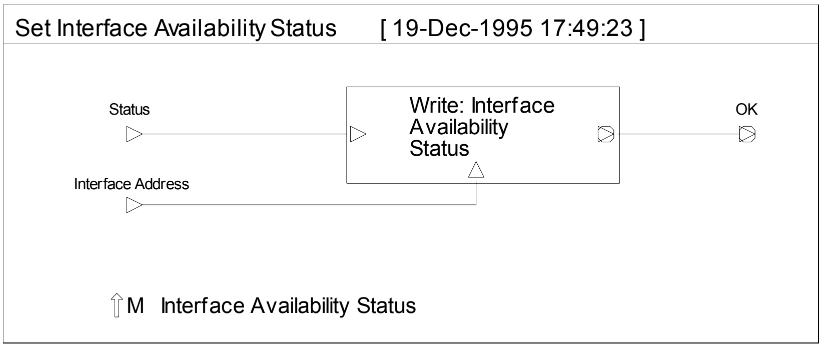
2.6.2.15. Get Interface Load Status



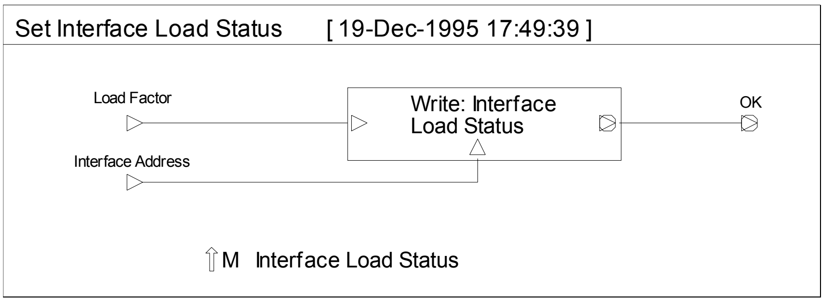
2.6.2.16. Get Routing Table Entry



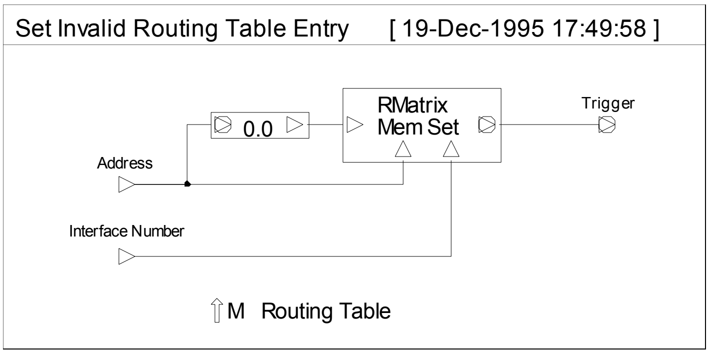
2.6.2.17. Set Interface Availability Status


Set Interface Availability Status [19-Dec-1995 17:49:23]
Status
Interface Address
Write: Interface Availability Status
OK
⇑M Interface Availability Status

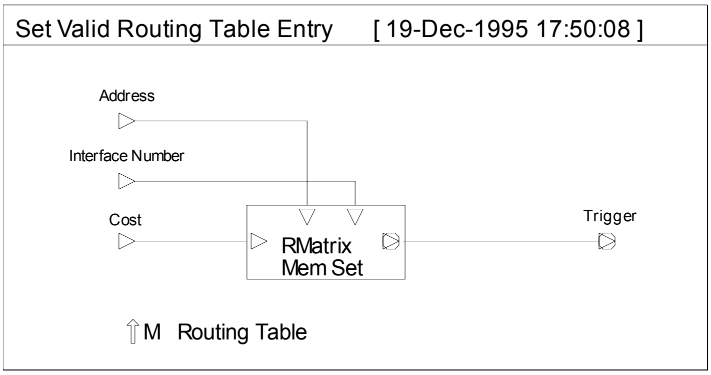
2.6.2.18. Set Interface Load Status


Set Interface Load Status [19-Dec-1995 17:49:39]
Load Factor
Interface Address
Write: Interface Load Status
OK
⇑M Interface Load Status

2.6.2.19. Set Invalid Routing Table Entry

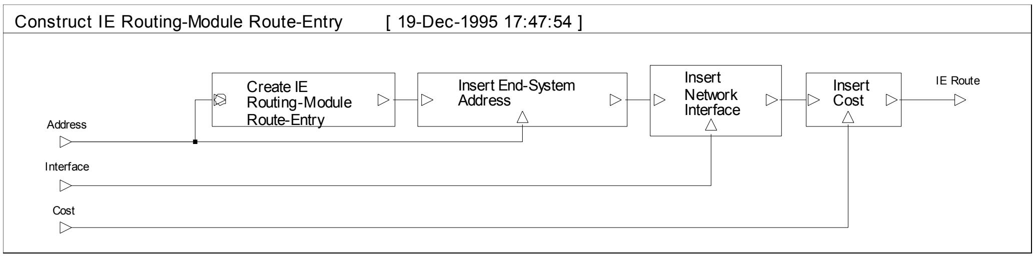


2.6.2.20. Set Valid Routing Table Entry

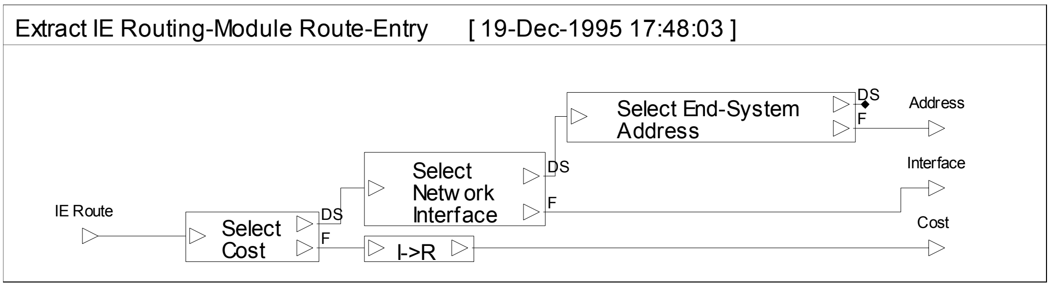


2.6.3. Support Modules

2.6.3.1. Construct IE Routing-Module Route Entry


Construct IE Routing-Module Route-Entry [19-Dec-1995 17:47:54]

2.6.3.2. Extract IE Routing-Module Route Entry


Extract IE Routing-Module Route-Entry [19-Dec-1995 17:48:03]

2.7. Generator

2.7.1. Data Structures

2.7.1.1. IE Generator Primitive

This Data Structure has no content.

2.7.1.2. IE Generator Setup-Primitive

Name	*Type*	*Subrange*	*Default Value*
Maximum Time	REAL	(0,+Inf)	1.0E9
Maximum Byte Count	INTEGER	(0,+Inf)	1000000000
Maximum Element Count	INTEGER	(0,+Inf	1000000000

2.7.1.3. IE Generator Setup-FTP

Name	*Type*	*Subrange*	*Default Value*
Maximum Time	*REAL*	*(0,+Inf)*	*1.0E9*
Maximum Byte Count	*INTEGER*	*(0,+Inf)*	*1000000000*
Maximum Element Count	*INTEGER*	*(0,+Inf)*	*1000000000*

2.7.1.4. IE Generator Setup-Statistical

Name	*Type*	*Subrange*	*Default Value*
Maximum Time	*REAL*	*(0,+Inf)*	*1.0E9*
Maximum Byte Count	*INTEGER*	*(0,+Inf)*	*1000000000*
Maximum Element Count	*INTEGER*	*(0,+Inf)*	*1000000000*
Time Characteristic	Statistical Parameter		
Space Characteristic	Statistical Parameter		

2.7.1.5. IE Generator Setup-Telnet

Name	*Type*	*Subrange*	*Default Value*
Maximum Time	*REAL*	*(0,+Inf)*	*1.0E9*
Maximum Byte Count	*INTEGER*	*(0,+Inf)*	*1000000000*
Maximum Element Count	*INTEGER*	*(0,+Inf)*	*1000000000*

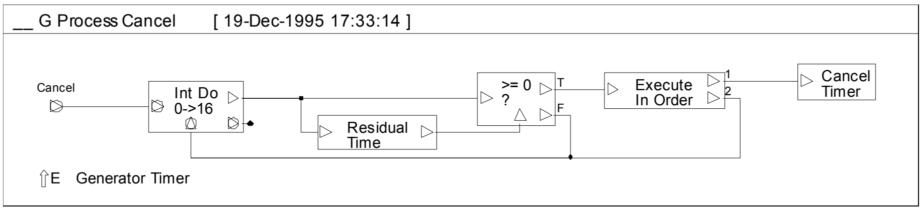
2.7.1.6. IE Generator Stop

This Data Structure has no content.

2.7.2. Main Modules

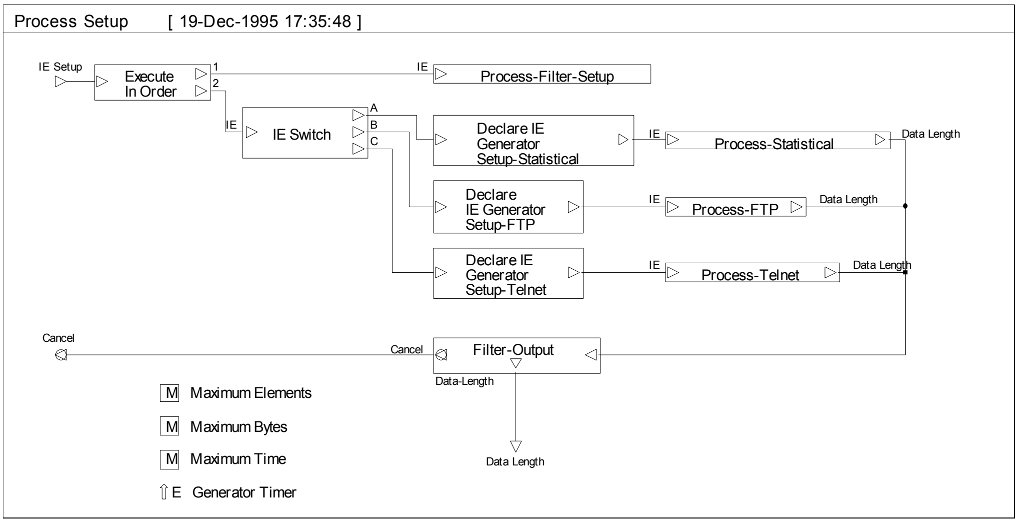
2.7.2.1. Process Cancel

This Module implements “PSPEC 1: Cancel Timers”.



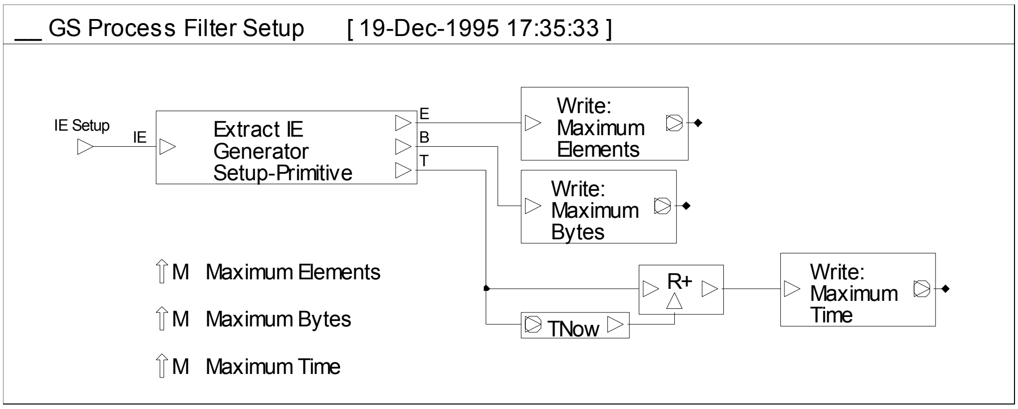
2.7.2.2. Process Setup

This Module implements “DFD 3: Setup Generator”. This also includes “PSPEC 3.1: Classify Type of Setup IE”.



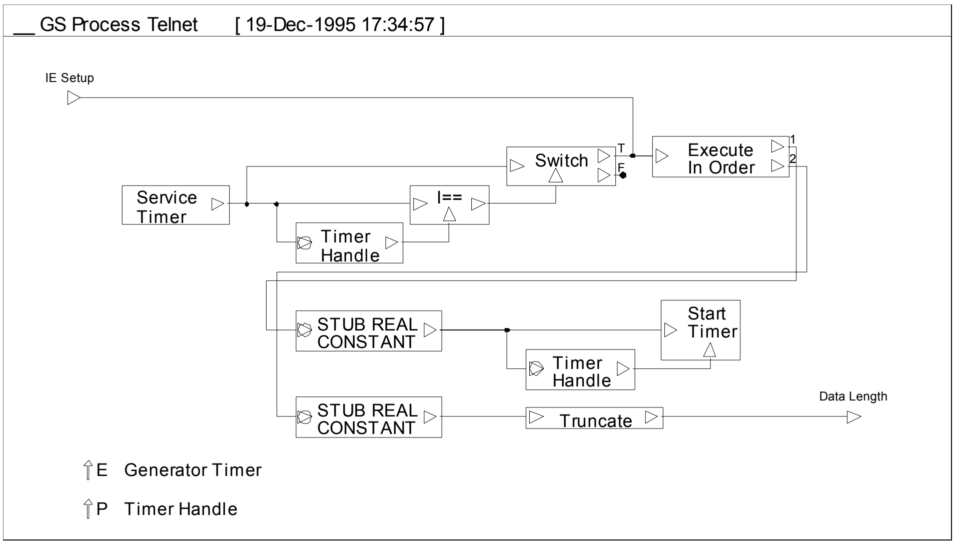
2.7.2.3. Process Setup -- Process Filter Setup

This Module implements “PSPEC 3.2: Setup Filter Parameters”.



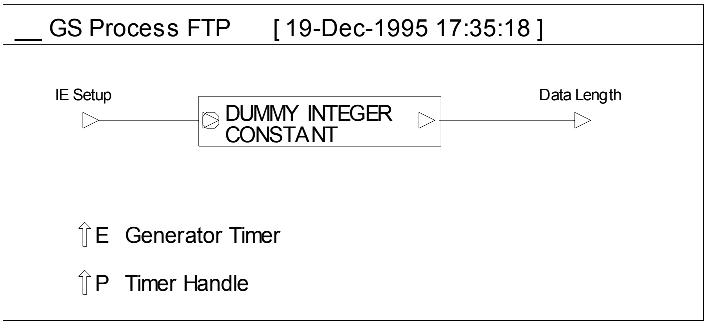
2.7.2.4. Process Setup -- Process Telnet

This Module implements “DFD 3.3: Telnet Processing”.



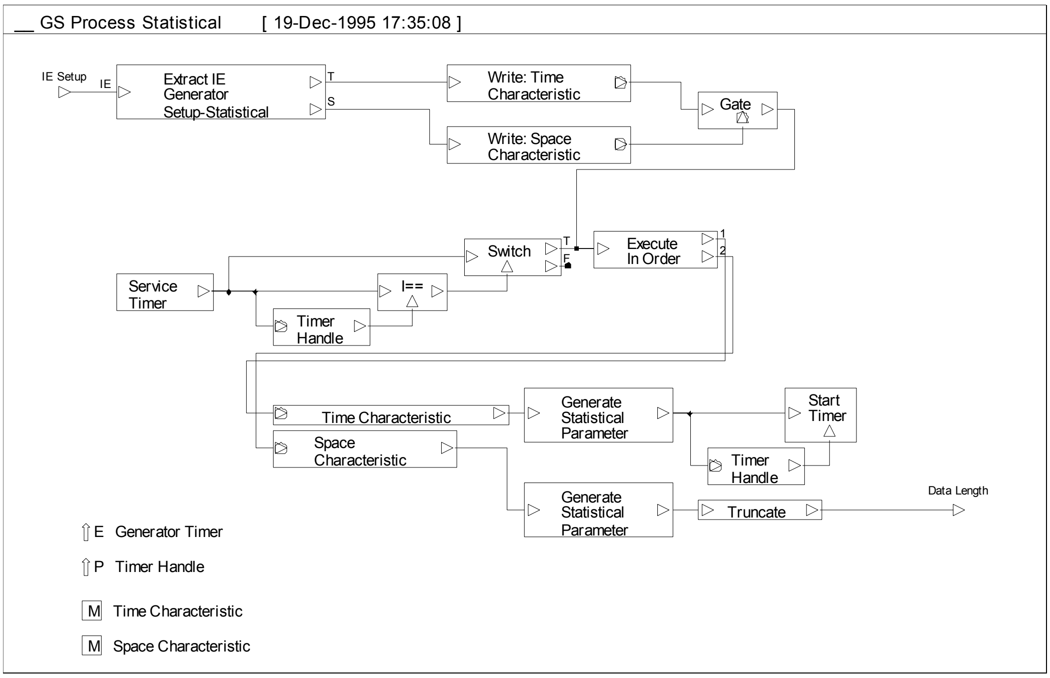
2.7.2.5. Process Setup -- Process FTP

This Module implements "DFD 3.4: FTP Processing".



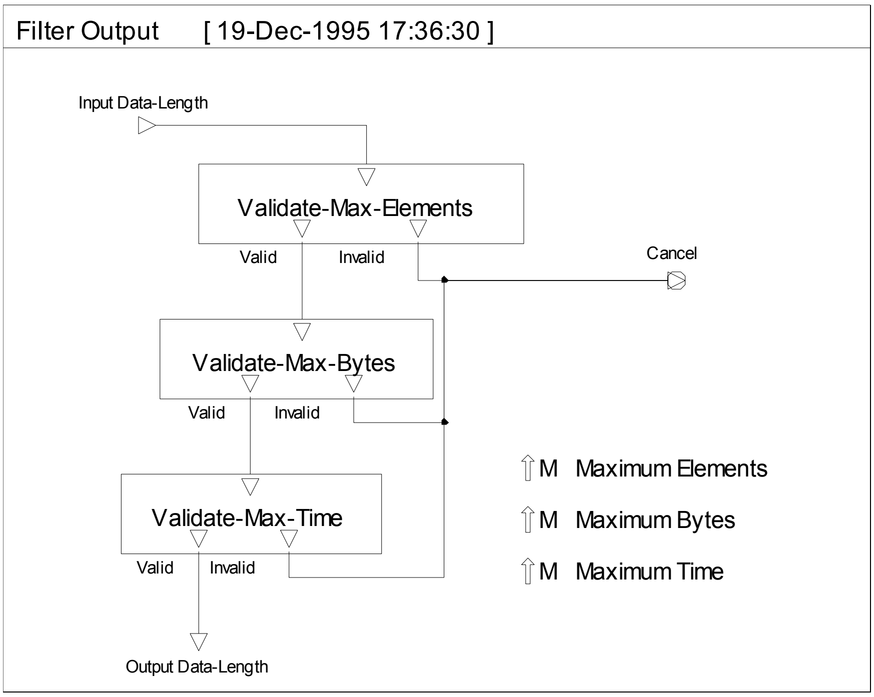
2.7.2.6. Process Setup -- Process Statistical

This Module implements "DFD 3.5: Statistical Processing".



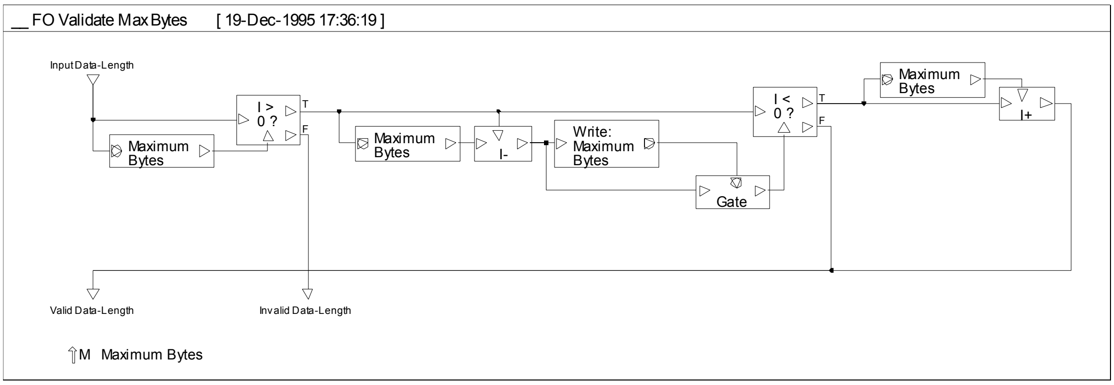
2.7.2.7. Process Setup -- Filter Output

This Module implements “PSPEC 3.6: Filter Output”.



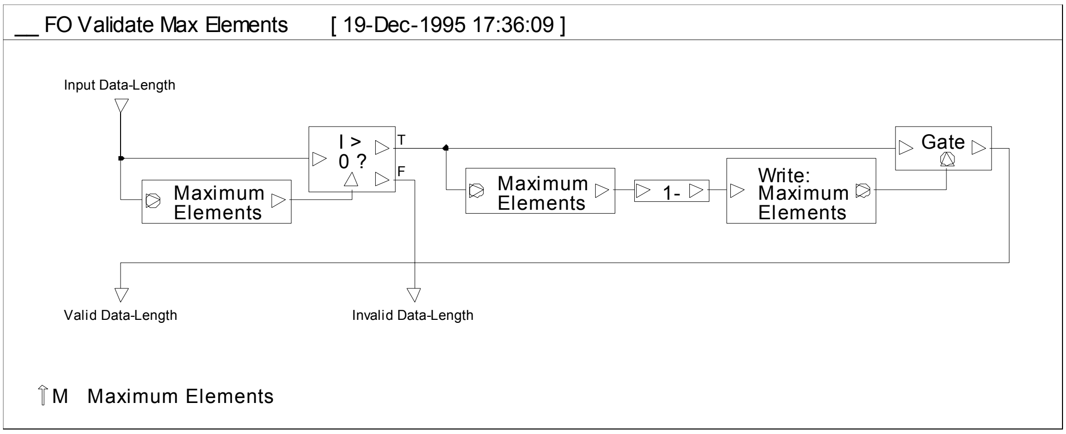
2.7.2.8. Process Setup -- Filter Output -- Validate Max Bytes

This Module implements “PSPEC 3.6: Filter Output”.



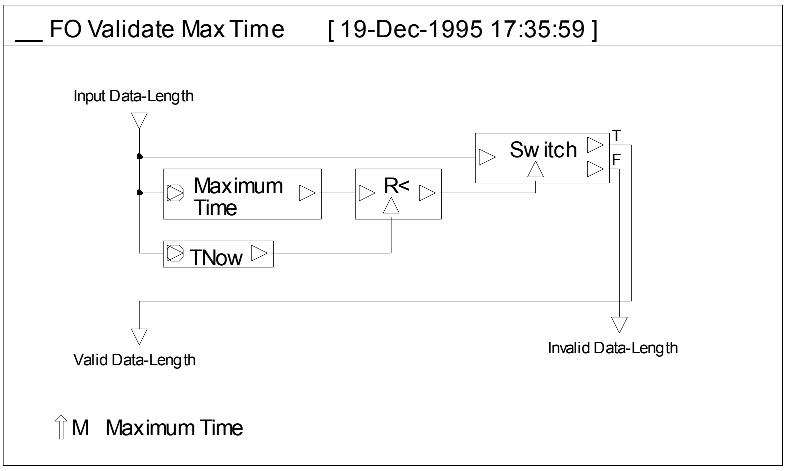
2.7.2.9. Process Setup -- Filter Output -- Validate Max Elements

This Module implements “PSPEC 3.6: Filter Output”.

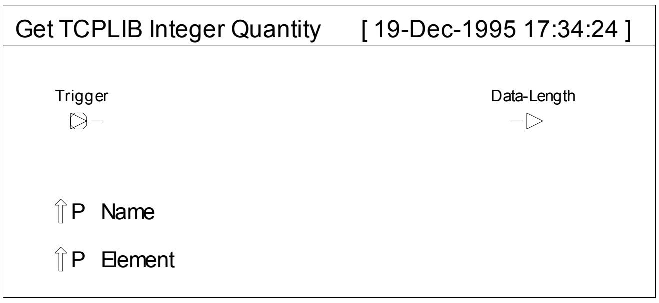


2.7.2.10. Process Setup -- Filter Output -- Validate Max Time

This Module implements “PSPEC 3.6: Filter Output”.



2.7.2.11. Get TCPLIB Integer Quantity

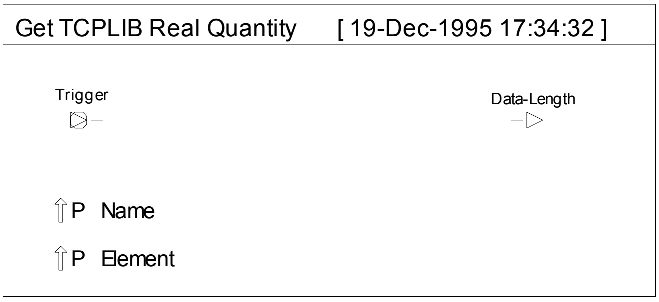


Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* --------------------------------------------------------------------- */
#   include "/u/mgream/BONeS/Constructed/TCPLib.c"
/* --------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User INIT Below Here */

/* --------------------------------------------------------------------- */
```

```
    /* 0. Data */
    char * profileId = Name;
    char * parameterId = Element;
    Function * Func = GetFunction (profileId, parameterId);

    /* 1. Seed the random number generator.
     */
    srand48 (time (NULL) ^ getpid ());

    /* 2. Verify that the supplied profileId and parameterId are
        correct; we only need to do this at the start of the
        simulation since the names are invariant.
     */
    if (Func == NULL)
      {
        char * modname = MODULE_NAMESTRING;
        char message[256];
        sprintf (message, "Cannot locate (%s, %s)", profileId, parameterId);
        __ReportError (modname, message);
      }
    else if (Func->Type != IntegerFunction)
      {
        char * modname = MODULE_NAMESTRING;
        char message[256];
        sprintf (message, "(%s, %s) is not a Integer type",
                profileId, parameterId);
        __ReportError (modname, message);
      }

    __Bfree (profileId);
    __Bfree (parameterId);
/* -------------------------------------------------------------------- */

    /* User INIT Above Here */

...

    /* User RUN Below Here */

/* -------------------------------------------------------------------- */
    /* 1. Locate the parameterIds for this particular TCPLIB quanity,
        by getting the profileId and parameterId names as they appear
        as arguments. Also attempt to load the function handler for
        this quantity.
     */
    char * profileId = Name;
    char * parameterId = Element;
    Function * Func = GetFunction (profileId, parameterId);

    /* 2. Free up resources
     */
    __Bfree (profileId);
    __Bfree (parameterId);
    __FreeArc (Trigger);

    /* 3. We considered errors already, so silent ignore
     */
    if (Func != NULL) {
        int IntValue = INT_FUNCTION (Func) ();
        __PutINTEGERVal (Data_Length, IntValue);
      }
/* -------------------------------------------------------------------- */

    /* User RUN Above Here */

```

2.7.2.12. Get TCPLIB Real Quantity

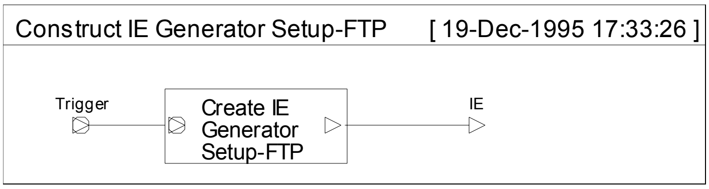


Extracts of the ‘C’ interface provided by BONeS are as follows.

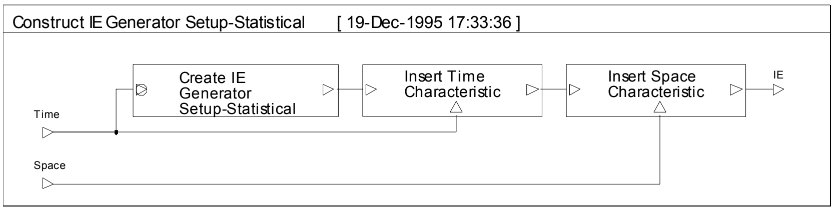
```

/* User GLOBAL-DEFINES Below Here */

/* -------------------------------------------------------------------- */
#   include "/u/mgream/BONeS/Constructed/TCPLib.c"
/* -------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User INIT Below Here */

/* -------------------------------------------------------------------- */
    /* 0. Data */
    char * profileId = Name;
    char * parameterId = Element;
    Function * Func = GetFunction (profileId, parameterId);

    /* 1. Seed the random number generator.
     */
    srand48 (time (NULL) ^ getpid ());

    /* 2. Verify that the supplied profileId and parameterId are
        correct; we only need to do this at the start of the
        simulation since the names are invariant.
     */
    if (Func == NULL)
      {
        char * modname = MODULE_NAMESTRING;
        char message[256];
        sprintf (message, "Cannot locate (%s, %s)", profileId, parameterId);
        __ReportError (modname, message);
      }
    else if (Func->Type != FloatFunction)
      {
        char * modname = MODULE_NAMESTRING;
        char message[256];
        sprintf (message, "(%s, %s) is not a Real type", profileId, parameterId);
        __ReportError (modname, message);
      }

      __Bfree (profileId);
      __Bfree (parameterId);
/* -------------------------------------------------------------------- */

    /* User INIT Above Here */

...

    /* User RUN Below Here */

/* -------------------------------------------------------------------- */
    /* 1. Locate the parameterIds for this particular TCPLIB quanity,
        by getting the profileId and parameterId names as they appear
        as arguments. Also attempt to load the function handler for
        this quantity.
     */
    char * profileId = Name;
    char * parameterId = Element;
    Function * Func = GetFunction (profileId, parameterId);

    /* 2. Free up resources
     */
    __Bfree (profileId);
    __Bfree (parameterId);
    __FreeArc (Trigger);

    /* 2. We considered errors already, so silent ignore
     */
    if (Func != NULL) {
        double Value = FLOAT_FUNCTION (Func) ();
        __PutREALVal (Data_Length, Value);
      }
/* -------------------------------------------------------------------- */

    /* User RUN Above Here */

```

2.7.3. Support Modules

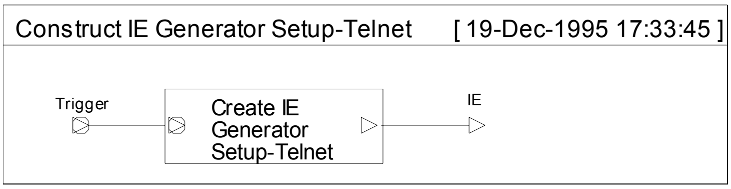
2.7.3.1. Construct IE Generator Setup FTP


Construct IE Generator Setup-FTP [19-Dec-1995 17:33:26]
Trigger
Create IE Generator Setup-FTP
IE

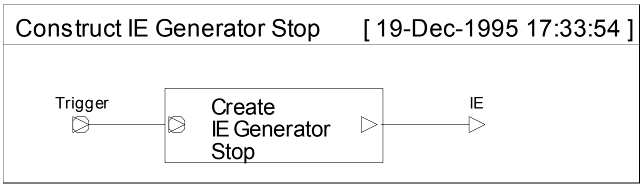
2.7.3.2. Construct IE Generator Setup Statistical


Construct IE Generator Setup-Statistical [19-Dec-1995 17:33:36]
Time
Space
Create IE Generator Setup-Statistical
Insert Time Characteristic
Insert Space Characteristic
IE

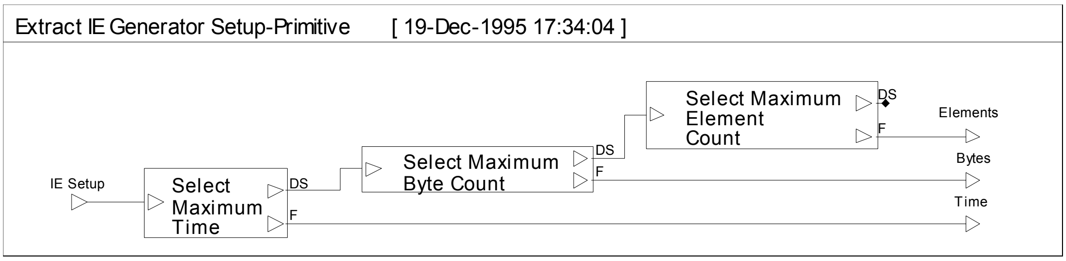
2.7.3.3. Construct IE Generator Setup Telnet


Construct IE Generator Setup-Telnet [19-Dec-1995 17:33:45]
Trigger
Create IE Generator Setup-Telnet
IE

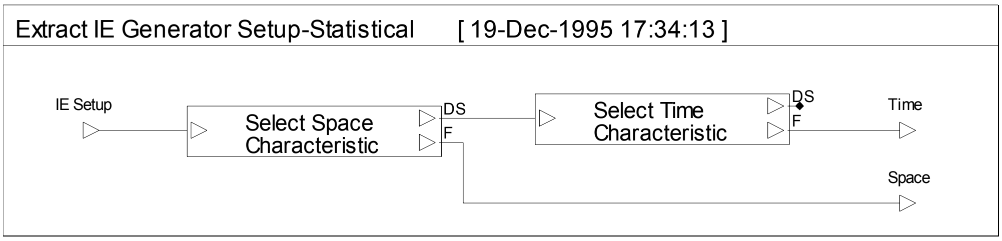
2.7.3.4. Construct IE Generator Stop



2.7.3.5. Extract IE Generator Setup Primitive


Extract IE Generator Setup-Primitive [19-Dec-1995 17:34:04]
IE Setup
Select Maximum Time
DS
F
Select Maximum Byte Count
DS
F
Select Maximum Element Count
DS
F
Elements
Bytes
Time

2.7.3.6. Extract IE Generator Setup Statistical


Extract IE Generator Setup-Statistical [19-Dec-1995 17:34:13]
IE Setup
Select Space Characteristic
DS
F
Select Time Characteristic
DS
F
Time
Space

2.7.4. ‘C’ Modules

There is a single interface module that bridges between the TCP Library itself, and the BONeS Primitive Modules. In addition, the TCP Library is provided as source code that compiles into a library. For the purposes of this project, the source code was concatenated into a single file and the source included -- this does mean that two separate instances of the TCP Library are compiled into the simulation, however this is an insignificant overhead.

2.7.4.1. TCP Library

```

/* ------------------------------------------------------------------- */
/* Module: TCPLIB
 * Filename: TCPLib.c
 * Author: Matthew Gream (90061060)
 * Description: provides a generic interface to lower level TCPLIB
 *  functions. specific functions can be obtained given a set of
 *  identifiers
 * RCS: $Id: TCPLib.c,v 1.1 1995/12/20 08:03:33 mgream Exp $
 */
/* ------------------------------------------------------------------- */
#ifdef TEST
#   include <stdio.h>
#   include <string.h>
#   include <time.h>
#endif

#ifndef TEST
#   define perror(msg)  __ReportError (MODULE_NAMESTRING, msg)
#endif
#   include "/u/mgream/BONeS/Constructed/tcplib/tcplib_src.c"

/* ------------------------------------------------------------------- */
extern long time ();
extern void srand48 ();
extern int getpid ();

/* ------------------------------------------------------------------- */
/* convert character into upper case */
static char my_toupper (ch)
    char ch;
  {
    if (ch >= 'a' && ch <= 'z')
        return (ch - 'a' + 'A');
    else
        return (ch);
  }

/* -------------------------------------------------------------------- */
/* compare without regard to case */
static int my_strcasecmp (stra, strb)
    char * stra;
    char * strb;
  {
    while (my_toupper (*stra) == my_toupper (*strb)) {
        if (*stra == '\0')
            return 0;
        stra++;
        strb++;
      }
    return 1;
  }

/* -------------------------------------------------------------------- */
/* define virtual functions :-) */
typedef enum {
    FloatFunction,
    IntegerFunction
  } FunctionType;
typedef int (*IntegerFunc) ();
typedef float (*FloatFunc) ();
typedef struct Function_ST {
    FunctionType Type;
    IntegerFunc Func;
  } Function;
#define INT_FUNCTION(f)     (*(*(IntegerFunc *)(&(f)->Func)))
#define FLOAT_FUNCTION(f)   (*(*(FloatFunc *)(&(f)->Func)))

/* -------------------------------------------------------------------- */
/* parameter type, and associated function */
typedef struct Parameter_ST {
    char * Ident;
    Function GetValue;
  } Parameter;

/* -------------------------------------------------------------------- */
/* profile type, and associated parameters */
typedef struct Profile_ST {
    char * Ident;
    Parameter * Parameters;
  } Profile;

/* -------------------------------------------------------------------- */
/* parameters for Telnet profile */
static Parameter Parameters_Telnet[] = {
    { "Packet Size", { IntegerFunction, telnet_pktsize } },
    { "Interarrival Time", { FloatFunction, telnet_interarrival } },
    { "Conversation Size", { FloatFunction, telnet_duration } },
    { NULL, { 0, NULL } },
  };

/* -------------------------------------------------------------------- */
/* parameters for FTP profile */
static Parameter Parameters_FTP[] = {
    { "Number Items", { IntegerFunction, ftp_nitems } },
    { "Item Size", { IntegerFunction, ftp_itemsize } },
    { "Control Size", { IntegerFunction, ftp_ctlsize } },
    { NULL, { 0, NULL } },
  };

/* -------------------------------------------------------------------- */
/* list of known profiles */
static Profile Profiles[] = {
    { "Telnet", Parameters_Telnet },
    { "FTP", Parameters_FTP },
    { NULL, NULL },
  };

/* -------------------------------------------------------------------- */
/* locate the list of parameters for a given profile */
static Parameter * GetParameterFromProfile (Ident)
    char * Ident;
  {
    Profile * Prof;
    for (Prof = Profiles; Prof->Ident != NULL; Prof++) {
        if (my_strcasecmp (Prof->Ident, Ident) == 0)
            return Prof->Parameters;
      }
    return NULL;
  }

/* ------------------------------------------------------------------ */
/* locate a function call for the given set of parameters */
static Function * GetFunctionFromParameter (Param, Ident)
    Parameter * Param;
    char * Ident;
  {
    for (; Param->Ident != NULL; Param++) {
        if (my_strcasecmp (Param->Ident, Ident) == 0)
            return &Param->GetValue;
      }
    return NULL;
  }

/* ------------------------------------------------------------------ */
/* locate a function call for a given (profile, parameter) tuple */
static Function * GetFunction (ProfileIdent, ParameterIdent)
    char * ProfileIdent;
    char * ParameterIdent;
  {
    Parameter * Param = GetParameterFromProfile (ProfileIdent);
    if (Param == NULL)
        return NULL;
    return GetFunctionFromParameter (Param, ParameterIdent);
  }

/* ------------------------------------------------------------------ */
/* Test Code */

#ifdef TEST

void main (argc, argv)
    int argc;
    char ** argv;
  {
    Function * Func;

    srand48 ((long)(time (NULL) ^ getpid ()));

    printf (" (%s, %s) ==> ", argv[1], argv[2]);
    Func = GetFunction (argv[1], argv[2]);
    if (Func == NULL)
        printf ("Invalid\n");
    else
      {
        if (Func->Type == IntegerFunction)
            printf ("[int] %u\n", INT_FUNCTION (Func) ());
        else if (Func->Type == FloatFunction)
            printf ("[float] %f\n", FLOAT_FUNCTION (Func) ());
      }
  }

#endif

/* ------------------------------------------------------------------ */

```

2.8. Management

2.8.1. Data Structures

2.8.1.1. Msg Management Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

2.8.1.2. Msg Management Set Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Parameter	IE Primitive		
Address	INTEGER	[0,512)	0

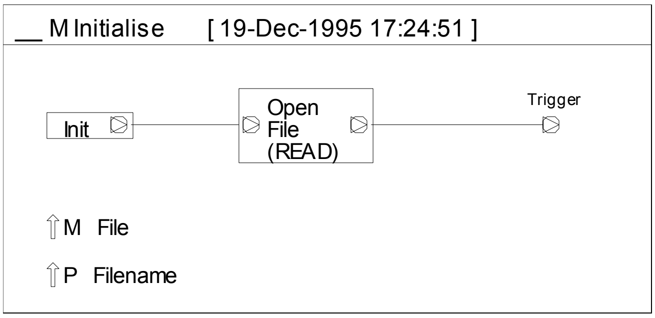
2.8.1.3. Msg Management Set Indication

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*
Parameter	IE Primitive		
Address	INTEGER	[0,512)	0

2.8.2. Main Modules

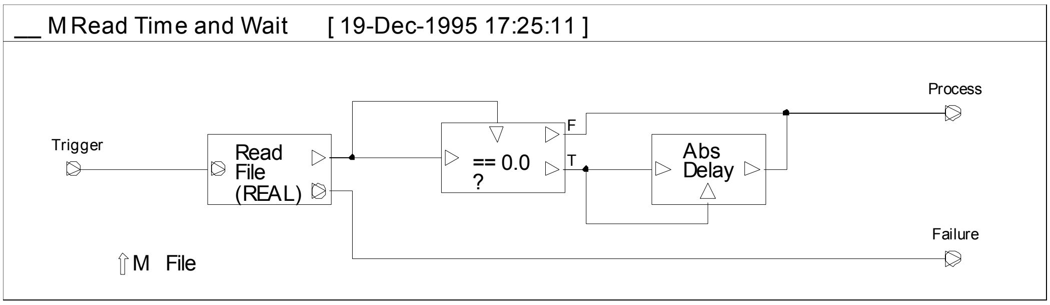
2.8.2.1. Initialise

This Module implements "PSPEC 6: Open and Initialise".



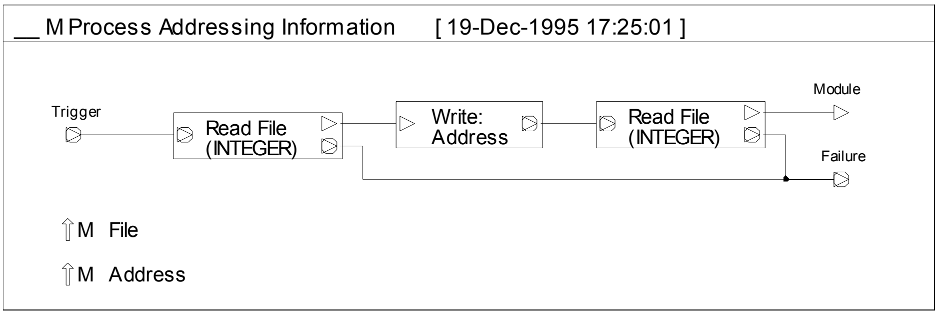
2.8.2.2. Read Time and Wait

This Module implements "PSPEC 1: Read and Wait for Next Entry".



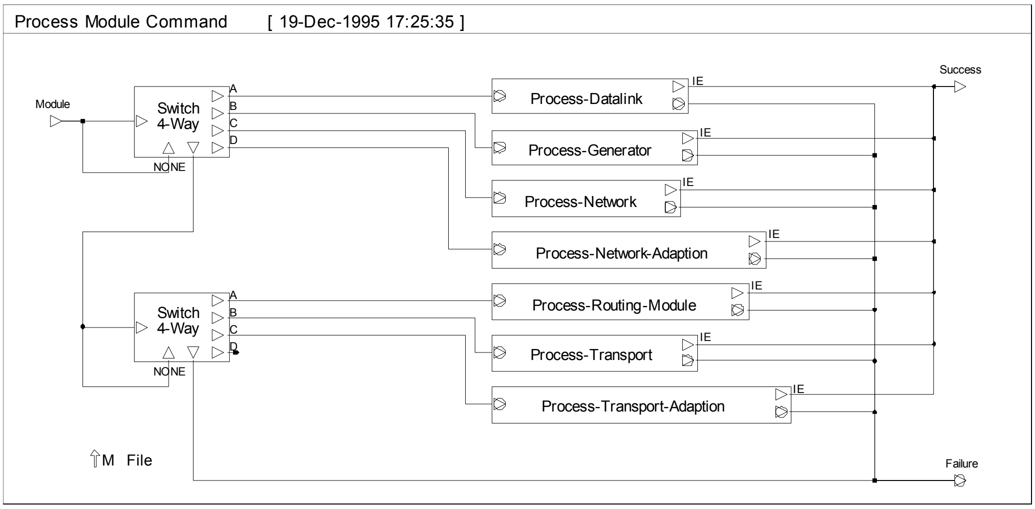
2.8.2.3. Process Addressing Information

This Module implements “PSPEC 2: Extract Address and Module”.



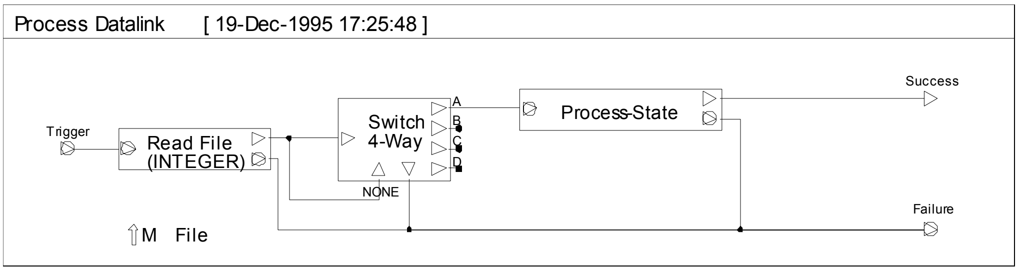
2.8.2.4. Process Module Command

This Module implements “DFD 3: Generate Specific IE”.



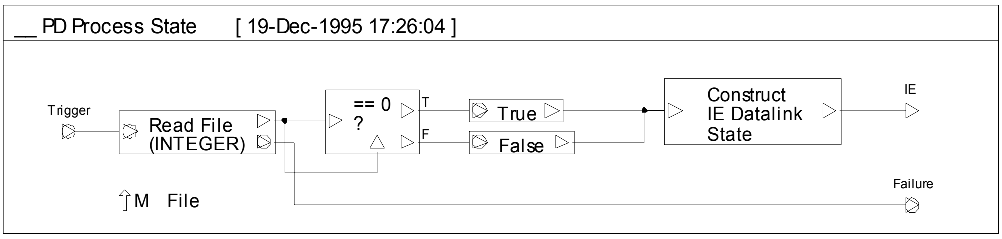
2.8.2.5. Process Module Command -- Process Datalink

This Module implements “PSPEC 3.7: Process Datalink IE”.



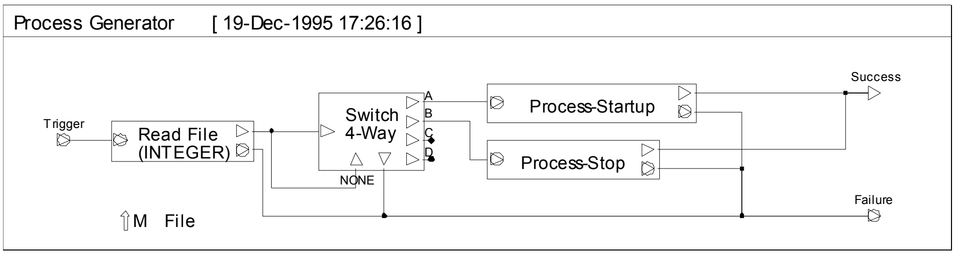
2.8.2.6. Process Module Command -- Process Datalink -- Process State

This Module implements “FUNCTION 3.7.1: Process State IE”.


__ PD Process State [19-Dec-1995 17:26:04]
Trigger
Read File (INTEGER)
== 0 ?
T
F
True
False
Construct IE Datalink State
IE
Failure
M File

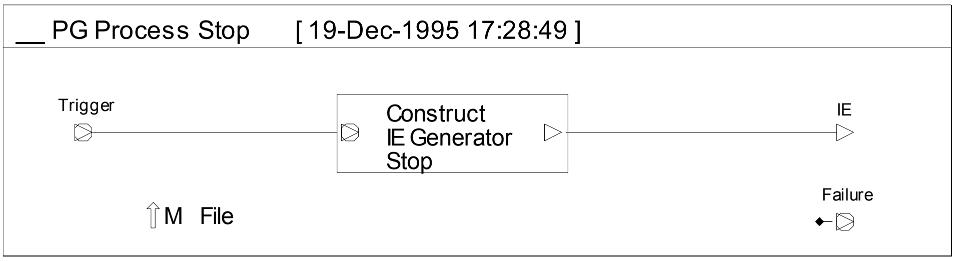
2.8.2.7. Process Module Command -- Process Generator

This Module implements “PSPEC 3.8: Process Generator IE”.



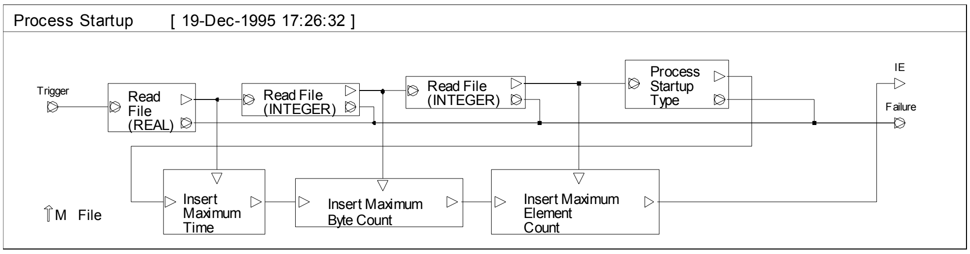
2.8.2.8. Process Module Command -- Process Generator -- Process Stop

This Module implements “FUNCTION 3.8.1: Process Stop IE”.


__ PG Process Stop [19-Dec-1995 17:28:49]
Trigger
Construct IE Generator Stop
IE
Failure
M File

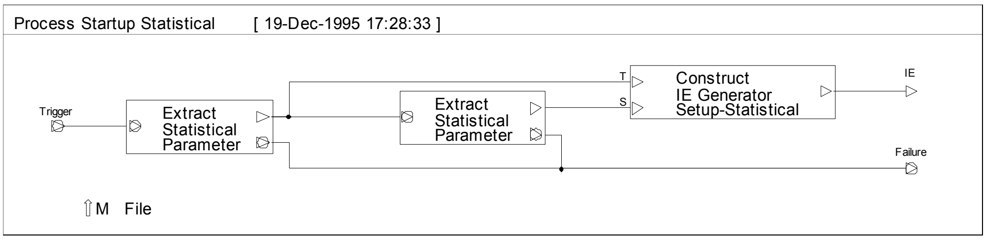
2.8.2.9. Process Module Command -- Process Generator -- Process Startup

This Module implements “FUNCTION 3.8.2: Process Setup IE”.



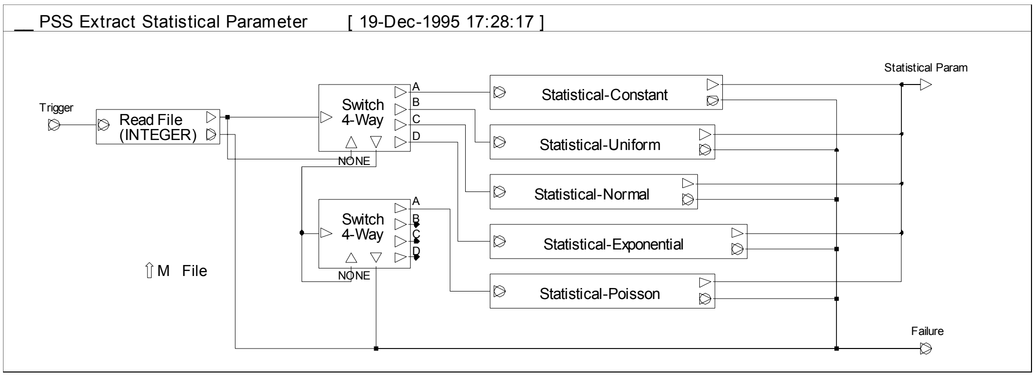
2.8.2.10. Process Module Command -- Process Generator -- Process Startup -- Process Startup Statistical

This Module implements “FUNCTION 3.8.3: Process Stat Info”.


Process Startup Statistical [19-Dec-1995 17:28:33]
Trigger
Extract Statistical Parameter
Extract Statistical Parameter
T
S
Construct IE Generator Setup-Statistical
IE
Failure
M File

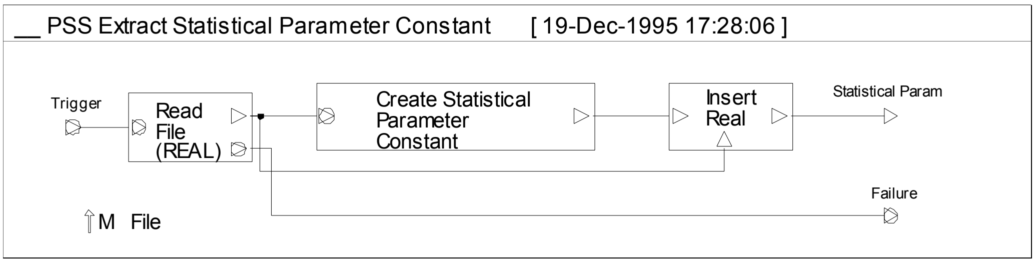
2.8.2.11. Process Module Command -- Process Generator -- Process Startup -- Process Startup Statistical -- Extract Statistical Parameter

This Module implements “FUNCTION 3.8.3: Process Stat Info”.



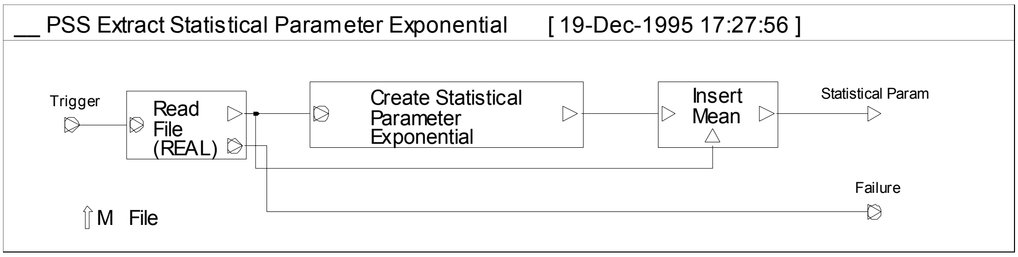
2.8.2.12. Process Module Command -- Process Generator -- Process Startup -- Process Startup Statistical -- Extract Statistical Parameter Constant

This Module implements "FUNCTION 3.8.3: Process Stat Info".


__ PSS Extract Statistical Parameter Constant [19-Dec-1995 17:28:06]
Trigger
Read File (REAL)
Create Statistical Parameter Constant
Insert Real
Statistical Param
Failure
M File

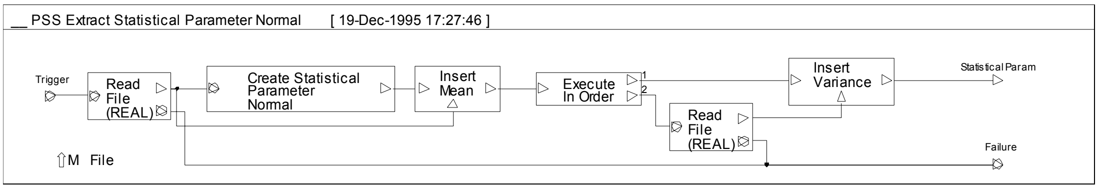
2.8.2.13. Process Module Command -- Process Generator -- Process Startup -- Process Startup Statistical -- Extract Statistical Parameter Exponential

This Module implements "FUNCTION 3.8.3: Process Stat Info".


__ PSS Extract Statistical Parameter Exponential [19-Dec-1995 17:27:56]
Trigger
Read File (REAL)
Create Statistical Parameter Exponential
Insert Mean
Statistical Param
Failure
M File

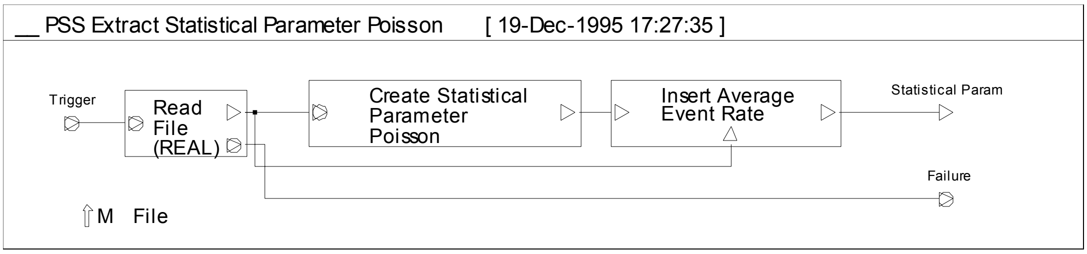
2.8.2.14. Process Module Command -- Process Generator -- Process Startup -- Process Startup Statistical -- Extract Statistical Parameter Normal

This Module implements "FUNCTION 3.8.3: Process Stat Info".



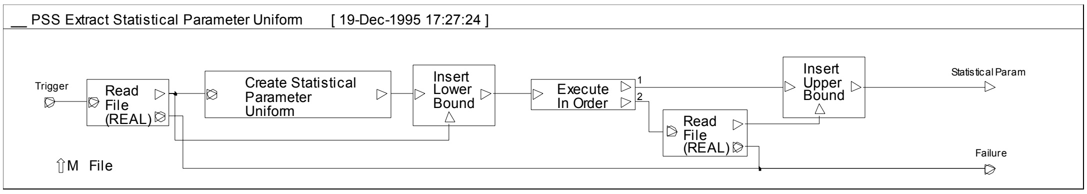
2.8.2.15. Process Module Command -- Process Generator -- Process Startup -- Process Startup Statistical -- Extract Statistical Parameter Poisson

This Module implements "FUNCTION 3.8.3: Process Stat Info".


__ PSS Extract Statistical Parameter Poisson [19-Dec-1995 17:27:35]
Trigger
Read File (REAL)
Create Statistical Parameter Poisson
Insert Average Event Rate
Statistical Param
Failure
M File

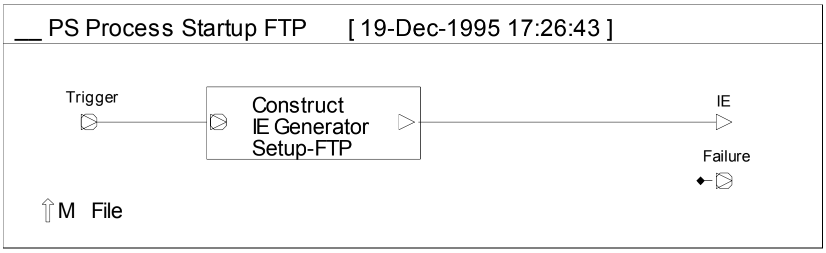
2.8.2.16. Process Module Command -- Process Generator -- Process Startup -- Process Startup Statistical -- Extract Statistical Parameter Uniform

This Module implements “FUNCTION 3.8.3: Process Stat Info”.



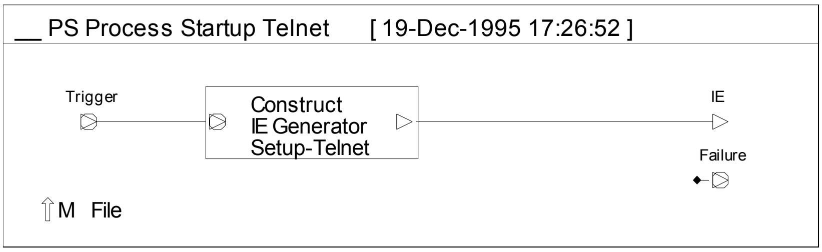
2.8.2.17. Process Module Command -- Process Generator -- Process Startup -- Process Startup FTP

This Module implements “FUNCTION 3.8.2: Process Setup IE”.


__ PS Process Startup FTP [19-Dec-1995 17:26:43]
Trigger
Construct IE Generator Setup-FTP
IE
Failure
M File

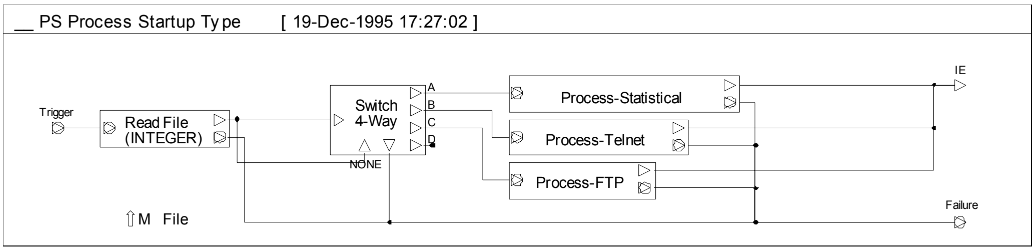
2.8.2.18. Process Module Command -- Process Generator -- Process Startup -- Process Startup Telnet

This Module implements “FUNCTION 3.8.2: Process Setup IE”.


__ PS Process Startup Telnet [19-Dec-1995 17:26:52]
Trigger
Construct IE Generator Setup-Telnet
IE
Failure
M File

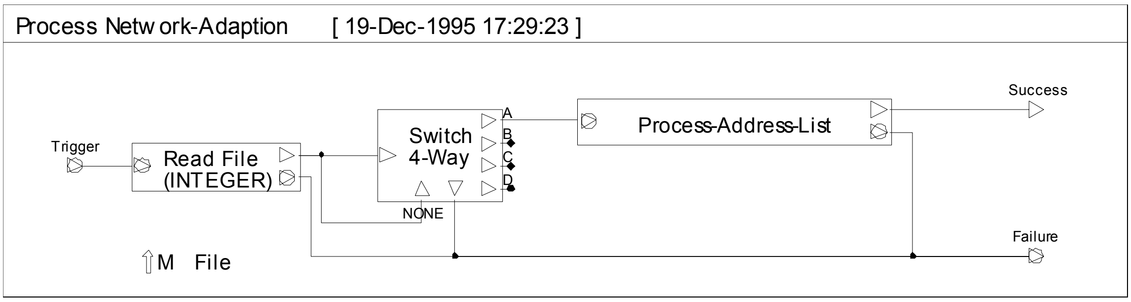
2.8.2.19. Process Module Command -- Process Generator -- Process Startup -- Process Startup Type

This Module implements “FUNCTION 3.8.2: Process Setup IE”.



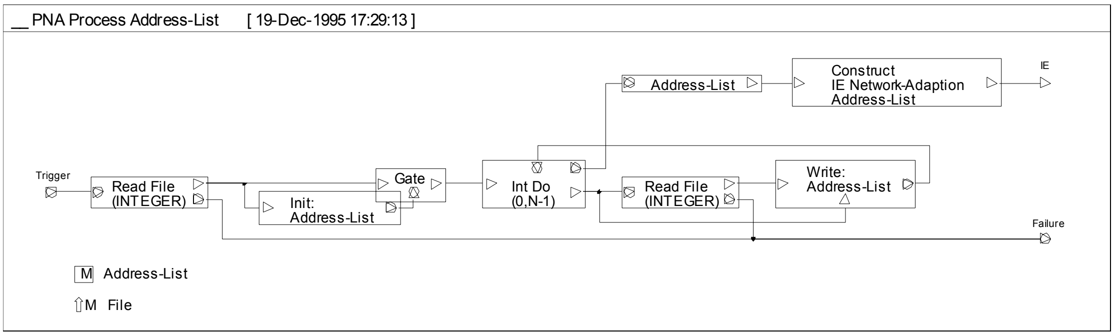
2.8.2.20. Process Module Command -- Process Network-Adaption

This Module implements “PSPEC 3.3: Process Network-Adaption IE”.



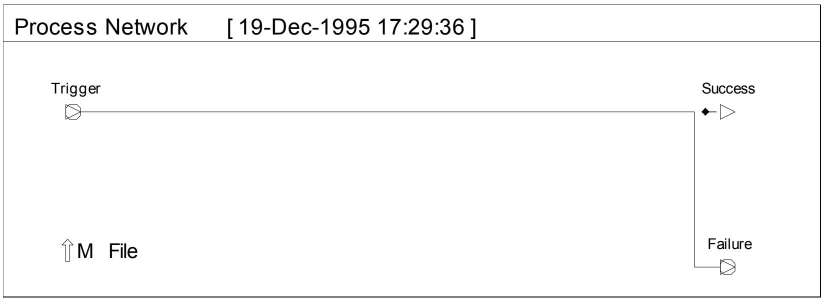
2.8.2.21. Process Module Command -- Process Network-Adaption -- Process Network Address List

This Module implements “FUNCTION 3.3.1: Process Address List IE”.



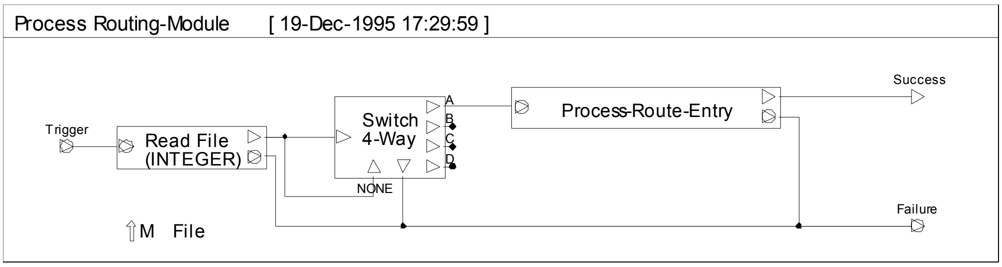
2.8.2.22. Process Module Command -- Process Network

This Module implements “PSPEC 3.4: Process Network IE”.



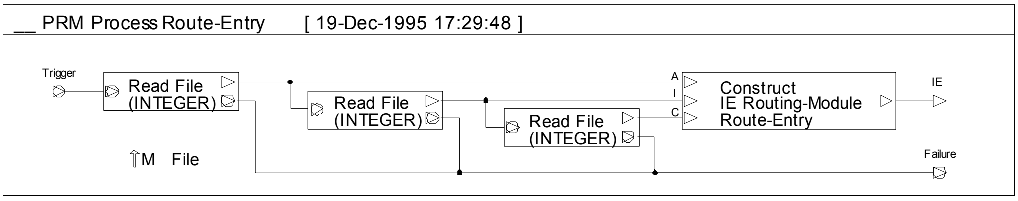
2.8.2.23. Process Module Command -- Process Routing-Module

This Module implements “PSPEC 3.6: Process Routing-Module IE”.



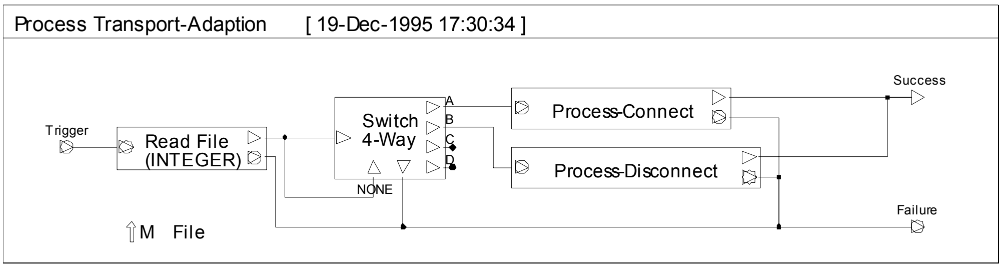
2.8.2.24. Process Module Command -- Process Routing-Module -- Process Route Entry

This Module implements “FUNCTION 3.6.1: Process Routing Entry IE”.



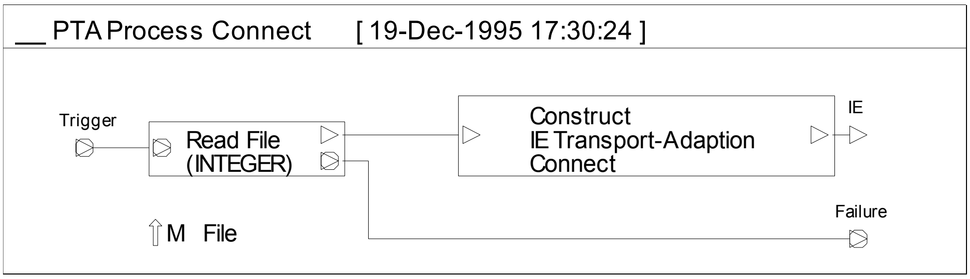
2.8.2.25. Process Module Command -- Process Transport-Adaption

This Module implements “PSPEC 3.2: Process Transport-Adaption IE”.



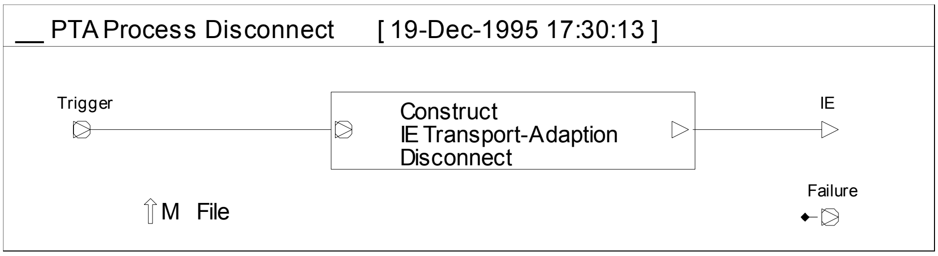
2.8.2.26. Process Module Command -- Process Transport-Adaption -- Process Connect

This Module implements “FUNCTION 3.2.1: Process Connect IE”.



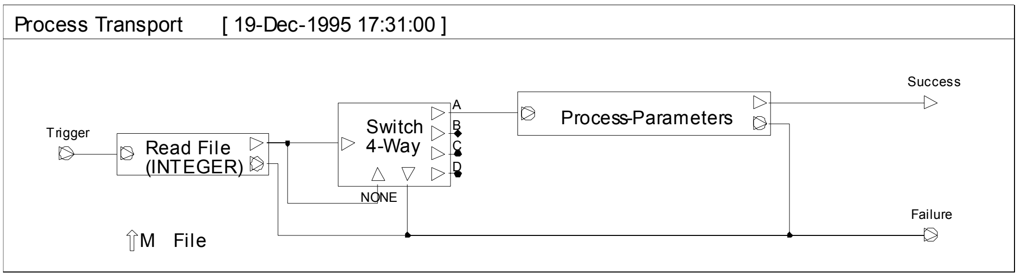
2.8.2.27. Process Module Command -- Process Transport-Adaption -- Process Disconnect

This Module implements “FUNCTION 3.2.2: Process Disconnect IE”.



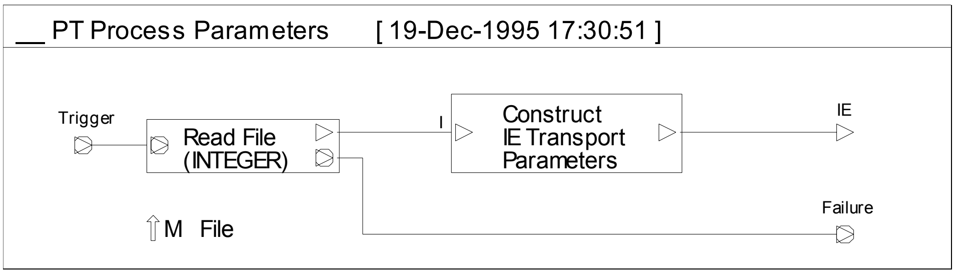
2.8.2.28. Process Module Command -- Process Transport

This Module implements "PSPEC 3.5: Process Transport IE".



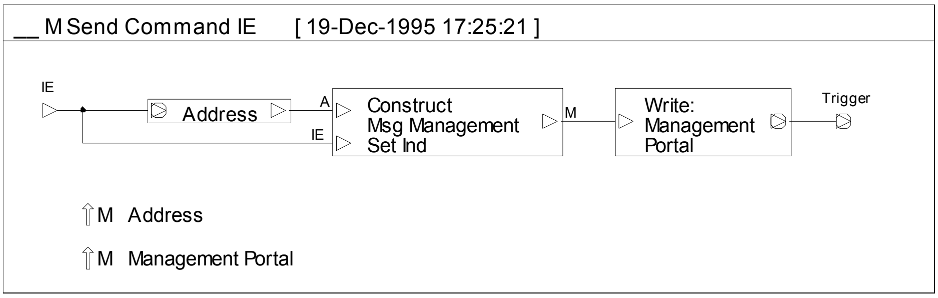
2.8.2.29. Process Module Command -- Process Transport -- Process Parameters

This Module implements "FUNCTION 3.5.1: Process Setup IE".



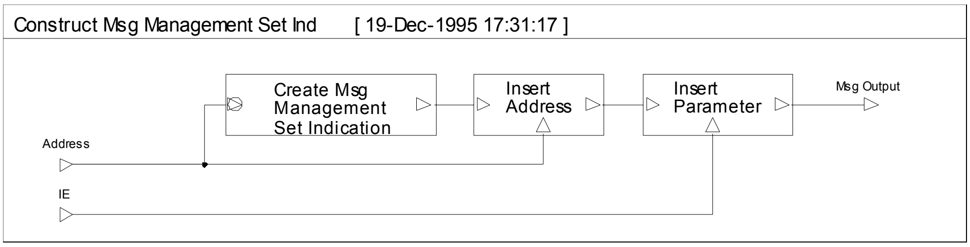
2.8.2.30. Send Command IE

This Module implements "PSPEC 4: Construct and Send Message".

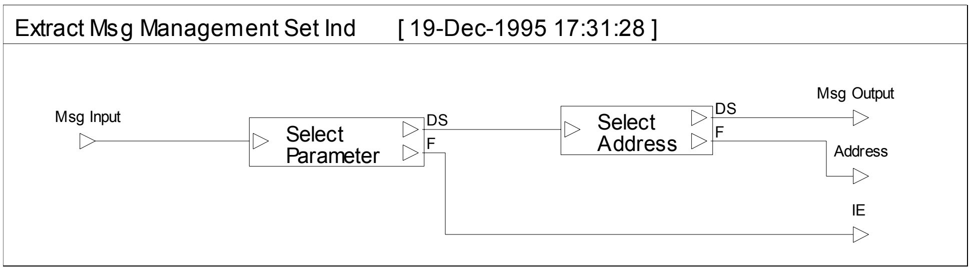


2.8.3. Support Modules

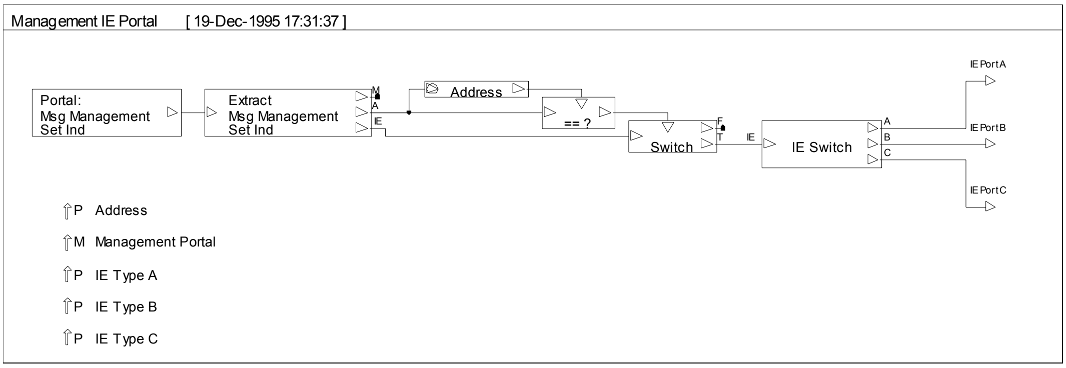
2.8.3.1. Construct Msg Management Set Indication


Construct Msg Management Set Ind [19-Dec-1995 17:31:17]
Create Msg Management Set Indication
Insert Address
Insert Parameter
Msg Output
Address
IE

2.8.3.2. Extract Msg Management Set Indication



2.8.3.3. Management IE Portal



3. Miscellaneous Modules

3.1. Statistical Parameter

3.1.1. Data Structures

3.1.1.1. Statistical Parameter

This Data Structure has no content.

3.1.1.2. Statistical Parameter Constant

Name	*Type*	*Subrange*	*Default Value*
Real	REAL	(-Inf,+Inf)	0.0

3.1.1.3. Statistical Parameter Exponential

Name	*Type*	*Subrange*	*Default Value*
Mean	REAL	(-Inf,+Inf)	0.0

3.1.1.4. Statistical Parameter Normal

Name	*Type*	*Subrange*	*Default Value*
Mean	REAL	(-Inf,+Inf)	0.0
Variance	REAL	(-Inf,+Inf)	0.0

3.1.1.5. Statistical Parameter Poisson

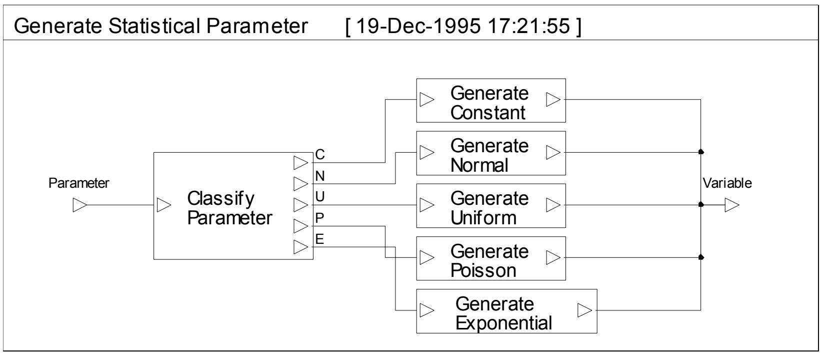
Name	*Type*	*Subrange*	*Default Value*
Average Event Rate	REAL	(-Inf,+Inf)	0.0

3.1.1.6. Statistical Parameter Uniform

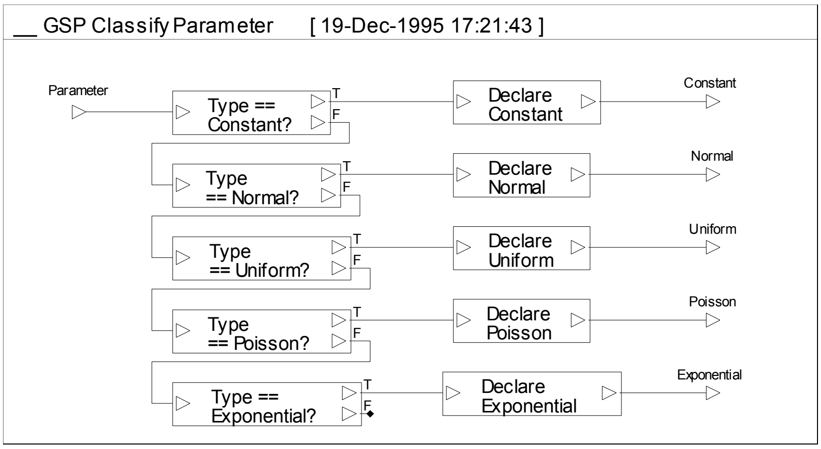
	Type	*Subrange*	*Default Value*
Upper Bound	REAL	(-Inf,+Inf)	1.0
Lower Bound	REAL	(-Inf,+Inf)	0.0

3.1.2. Modules

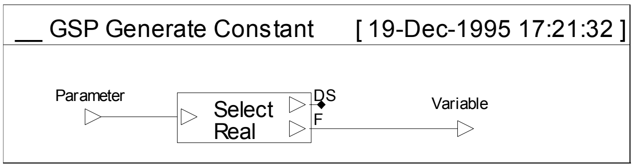
3.1.2.1. Generate Statistical Parameter



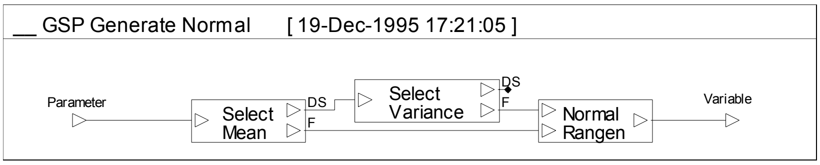
3.1.2.2. Generate Statistical Parameter -- Classify Parameter



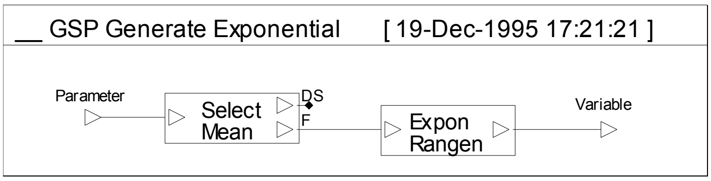
3.1.2.3. Generate Statistical Parameter -- Generate Constant



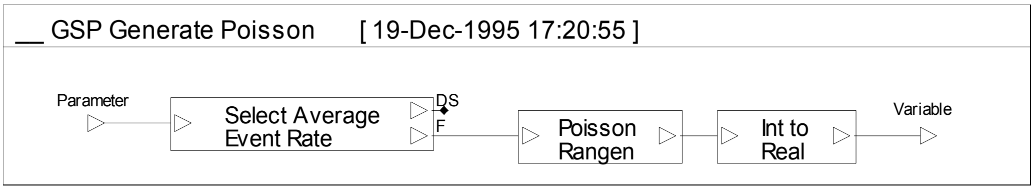
3.1.2.4. Generate Statistical Parameter -- Generate Normal



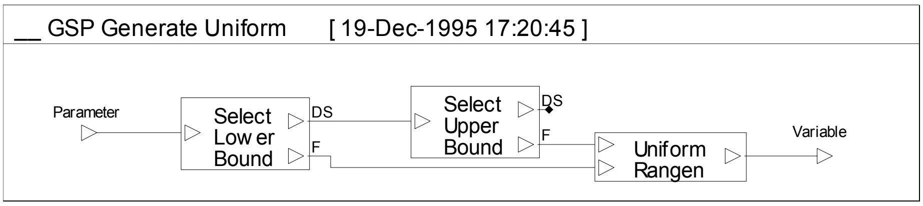
3.1.2.5. Generate Statistical Parameter -- Generate Exponential



3.1.2.6. Generate Statistical Parameter -- Generate Poisson


__ GSP Generate Poisson [19-Dec-1995 17:20:55]
Parameter
Select Average Event Rate
DS
F
Poisson Rangen
Int to Real
Variable

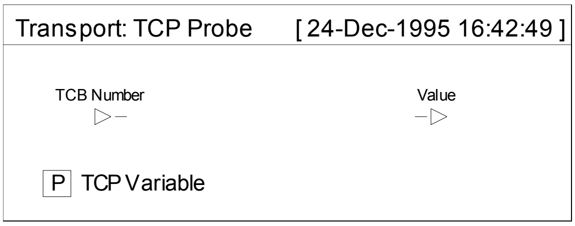
3.1.2.7. Generate Statistical Parameter -- Generate Uniform



3.2. Transport - TCP Probe

3.2.1. Modules

3.2.1.1. TCP Probe



Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* -------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Probes/TCP.c"
/* -------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User INIT Below Here */

/* -------------------------------------------------------------------- */
    BONeS_TCP_Probe_Init (argvector);
/* -------------------------------------------------------------------- */


    /* User INIT Above Here */

...

    /* User RUN Below Here */

/* -------------------------------------------------------------------- */
    BONeS_TCP_Probe_Execute (TCBNumber, Value, argvector);
/* -------------------------------------------------------------------- */

    /* User RUN Above Here */

```

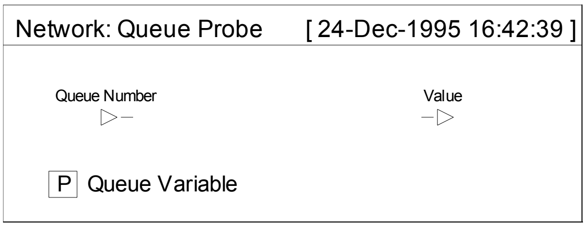
3.2.2. ‘C’ Modules

The Probe implementation uses a single ‘C’ module that interacts with the TCP Modules.

```

/* -------------------------------------------------------------------- */
/* $Id$
 * $Log$
 */
/* -------------------------------------------------------------------- */

#   include     "/u/mgream/BONeS/Constructed/TCP/TCP.c"

/* -------------------------------------------------------------------- */
/* PROBE COMPUTATION FUNCTIONS
 |
 | The following functions compute the necessary values from the Tcb.
 |
 */
static int Get_Congestion_Window (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = (double) Tcb->snd_cwnd;
    return 1;
  }
static int Get_Slow_Start_Threshold (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = (double) Tcb->snd_ssthresh;
    return 1;
  }
static int Get_ReTx_Events (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    double  retx_count = Tcb->PROBE_retx_count;
    Tcb->PROBE_retx_count = 0;
    if (retx_count == 0)
        return 0;
    (*value) = (double) retx_count;
    return 1;
  }
static int Get_RTT_Average (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = (double) Tcb->t_srtt;
    return 1;
  }
static int Get_RTT_Variance (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = (double) Tcb->t_rttvar;
    return 1;
  }
static int Get_Send_Window (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = (double) Tcb->snd_wnd;
    return 1;
  }
static int Get_Unacknowledged_Data (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = (double) Tcb->snd_wnd - (Tcb->snd_nxt - Tcb->snd_una);
    return 1;
  }
static int Get_Timer_Expiries (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    double  timer_exp  = Tcb->PROBE_timer_exp;
    Tcb->PROBE_timer_exp = 0;
    if (timer_exp == 0)
        return 0;
    (*value) = (double) timer_exp;
    return 1;
  }
static int Get_Ack_Received (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    double  ack_recv   = Tcb->PROBE_ack_recv;
    Tcb->PROBE_ack_recv = 0;
    if (ack_recv = 0)
        return 0;
    (*value) = (double) ack_recv;
    return 1;
  }
static int Get_KB_ReTx (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = (double) Tcb->PROBE_retx_count;
    return 1;
  }
static int Get_KB_Tx (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = (double) Tcb->PROBE_tx_count;
    return 1;
  }
static int Get_Reassembly_Queue_Size (Tcb, value)
    TcbPtr Tcb;
    double * value;
  {
    (*value) = QueueGetSize (Tcb->FragQueue);
    return 1;
  }

/* ------------------------------------------------------------------ */

/*  PROBE TABLE
 |
 |  The Table maintains an index of all the possible data that can
 |  be captured; each of which has a particular function that does
 |  the hard work.
 */

typedef double (*function) ();

typedef struct {
    char * string;
    function processor;
  } table_entry;

#   define              TCPProbeTableSize       11

static  table_entry     TCPProbeTable[TCPProbeTableSize] =
  {
    { "Congestion Window",            Get_Congestion_Window },
    { "Slow Start Threshold",         Get_Slow_Start_Threshold },
    { "Retransmission Events",        Get_ReTx_Events },
    { "Round Trip Time Average",      Get_RTT_Average },
    { "Round Trip Time Variance",     Get_RTT_Variance },
    { "Send Window",                  Get_Send_Window },
    { "Unacknowledged Data",          Get_Unacknowledged_Data },
    { "Timer Expiries",               Get_Timer_Expiries },
    { "Acknowledgements Received",    Get_Ack_Received },
    { "KB Retransmitted",             Get_KB_ReTx },
    { "KB Transmitted",               Get_KB_Tx },
    { "Reassembly Queue Size",        Get_Reassembly_Queue_Size },
  };

/* ------------------------------------------------------------------ */

/*  TCPProbeTableLookup
 |
 |  Given a particular string, look for the item in the table that we
 |  require.
 */
static int TCBProbeTableLookup (string)
    char * string;
  {
    int index;
    for (index = 0; index < TCPProbeTableSize; index++)
      {
        if (_strcasecmp (string, TCPProbeTable[index].string))
            return index;
      }
    return -1;
  }

/* ------------------------------------------------------------------ */

/*  TCPProbeTableCompute
 |
 |  Given the TCB and the particular datum that we require, go and do
 |  the computation using the function we've got configured in the
 |  table.
 */
static int TCPProbeTableCompute (Tcb, index, value)
    TcbPtr Tcb;
    int index;
    double * value;
  {
    return TCPProbeTable[index].processor (Tcb, value);
  }

/* -------------------------------------------------------------------- */
/* -------------------------------------------------------------------- */
/* -------------------------------------------------------------------- */

/*  BONeS_TCP_Probe_Init
 |
 |  Given the particular type of data that we need to look at,
 |  work through the table to locate the index, and therefore
 |  the function that will eventually serve us. Make a noise
 |  if the operation fails. Also, an indication is made that
 |  this is the "first value".
 */
static void BONeS_TCP_Probe_Init (argvector)
    arg_ptr argvector;
  {
    char * _DataType = __GetSTRINGVal (DataType_arc);
    int _TableIndex = TCPProbeTableLookup (_DataType);

    __PutINTEGERVal (FirstValue_arc, 0);
    __PutINTEGERVal (TableIndex_arc, _TableIndex);

    if (_TableIndex < 0)
      {
        __ReportError (MODULE_NAMESTRING, "Can't locate this Probe");
      }

    _Bfree (_DataType);
  }

/* -------------------------------------------------------------------- */

/*  BONeS_TCP_Probe_Execute
 |
 |  Runtime requires that we get the item of data, and then just output
 |  it if is it not different, depending on the duplicate parameter.
 */
static void BONeS_TCP_Probe_Execute (TCBNumber, ProbeOutput, argvector)
    arc_ptr TCBNumber;
    arc_ptr ProbeOutput;
    arg_ptr argvector;
  {
    TcbPtr Tcb = TcbLookup (__GetINTEGERVal (TCBNumber));
    int _TableIndex = __GetINTEGERVal (TableIndex_arc);

    if (Tcb != NULL && _TableIndex >= 0)
      {
        int     _Duplicate      = __GetINTEGERVal (Duplicate_arc);
        int     _First_Value    = __GetINTEGERVal (FirstValue_arc);
        double  _Old_Value      = __GetREALVal (OldValue_arc);
        double  _New_Value;

        if (TCPProbeTableCompute (Tcb, _TableIndex, &_New_Value) > 0 &&
                (_Duplicate == 0 ||
                 _New_Value != _Old_Value ||
                 _First_Value == 0))
          {
            __PutINTEGERVal (ProbeOutput, _New_Value);

            __PutINTEGERVal (FirstValue_arc, 1);
            __PutINTEGERVal (OldValue_arc, _New_Value);
          }

      }

    __FreeArc (TCBNumber);
  }

/* -------------------------------------------------------------------- */

```

3.2.2.1. Network - Queue Probe

3.2.3. Modules



Extracts of the ‘C’ interface provided by BONeS are as follows.

```

/* User GLOBAL-DEFINES Below Here */

/* ------------------------------------------------------------------- */
#   include     "/u/mgream/BONeS/Constructed/Probes/Queue.c"
/* ------------------------------------------------------------------- */

/* User GLOBAL-DEFINES Above Here */

...

    /* User INIT Below Here */

/* ------------------------------------------------------------------- */
    BONeS_Queue_Probe_Init (argvector);
/* ------------------------------------------------------------------- */

    /* User INIT Above Here */

...

    /* User RUN Below Here */

/* ------------------------------------------------------------------- */
    BONeS_Queue_Probe_Execute (QueueNumber, Value, argvector);
/* ------------------------------------------------------------------- */

    /* User RUN Above Here */

```

3.2.4. ‘C’ Modules

The Probe implementation uses a single ‘C’ module that interacts with the Queue Modules.

```

/* ------------------------------------------------------------------- */
/* $Id$
 * $Log$
 */
/* ------------------------------------------------------------------- */

#   include     "/u/mgream/BONeS/Constructed/Queue/Queue.c"

/* ------------------------------------------------------------------- */

static int Address;

/* ------------------------------------------------------------------- */
/* PROBE COMPUTATION FUNCTIONS
 |
 | The following functions compute the necessary values from the Queue.
 |
 */
```

```
static int Get_Size (QEntry, value)
    QueueEntry * QEntry;
    double * value;
  {
    (*value) = QueueSize (QEntry);
    return 1;
  }
static int Get_SrcAddressSize (QEntry, value)
    QueueEntry * QEntry;
    double * value;
  {
    int index;
    int size = QueueSize (QEntry->Que);

    (*value) = 0;
    for (index = 0; index < size; index++)
      {
        if (_Get_Src_Address (QueuePeekElement (QEntry->Que, index)) == Address)
            (*value)++;
      }

    return 1;
  }
static int Get_Size (QEntry, value)
    QueueEntry * QEntry;
    double * value;
  {
    int index;
    int size = QueueSize (QEntry->Que);

    (*value) = 0;
    for (index = 0; index < size; index++)
      {
        if (_Get_Dst_Address (QueuePeekElement (QEntry->Que, index)) == Address)
            (*value)++;
      }

    return 1;
  }
/* -------------------------------------------------------------------- */

/*  PROBE TABLE
 |
 |  The Table maintains an index of all the possible data that can
 |  be captured; each of which has a particular function that does
 |  the hard work.
 */

typedef double (*function) ();

typedef struct {
    char * string;
    function processor;
  } table_entry;

#   define              QueueProbeTableSize       3

static  table_entry     QueueProbeTable[QueueProbeTableSize] =
  {
    { "Size",                       Get_Size },
    { "Source Address Count",       Get_SrcAddressSize },
    { "Dest Address Count",         Get_DstAddressSize },
  };

/* -------------------------------------------------------------------- */

/*  QueueProbeTableLookup
 |
 |  Given a particular string, look for the item in the table that we
 |  require.
 */
static int QueueProbeTableLookup (string)
    char * string;
  {
    int index;
    for (index = 0; index < QueueProbeTableSize; index++)
      {
        if (_strcasecmp (string, QueueProbeTable[index].string))
            return index;
      }
    return -1;
  }

/* ------------------------------------------------------------------ */

/*  QueueProbeTableCompute
 |
 |  Given the Queue and the particular datum that we require, go and do
 |  the computation using the function we've got configured in the
 |  table.
 */
static int QueueProbeTableCompute (QEntry, index, address. value)
    QueueEntry * QEntry;
    int index;
    int address,
    double * value;
  {
    Address = address;
    return QueueProbeTable[index].processor (QEntry, value);
  }

/* ------------------------------------------------------------------ */
/* ------------------------------------------------------------------ */
/* ------------------------------------------------------------------ */

/*  BONeS_Queue_Probe_Init
 |
 |  Given the particular type of data that we need to look at,
 |  work through the table to locate the index, and therefore
 |  the function that will eventually serve us. Make a noise
 |  if the operation fails. Also, an indication is made that
 |  this is the "first value".
 */
static void BONeS_Queue_Probe_Init (argvector)
    arg_ptr argvector;
  {
    char * _DataType = __GetSTRINGVal (DataType_arc);
    int _TableIndex = QueueProbeTableLookup (_DataType);

    __PutINTEGERVal (FirstValue_arc, 0);
    __PutINTEGERVal (TableIndex_arc, _TableIndex);

    if (_TableIndex < 0)
      {
        __ReportError (MODULE_NAMESTRING, "Can't locate this Probe");
      }

    _Bfree (_DataType);
  }

/* ------------------------------------------------------------------ */

/*  BONeS_Queue_Probe_Execute
 |
 |  Runtime requires that we get the item of data, and then just output
 |  it if is it not different, depending on the duplicate parameter.
 */
static void BONeS_Queue_Probe_Execute (QueueNumber, ProbeOutput, argvector)
    arc_ptr QueueNumber;
    arc_ptr ProbeOutput;
    arg_ptr argvector;
  {
    int QIndex = __GetINTEGERVal (QueueNumber);
    QueueEntry * QEntry = &QueueTable[QIndex];
    int _TableIndex = __GetINTEGERVal (TableIndex_arc);

    if (QueueEntry != NULL && QIndex >= 0)
      {
        int     _Address        = __GetINTEGERVal (Address_arc);
        int     _Duplicate      = __GetINTEGERVal (Duplicate_arc);
        int     _First_Value    = __GetINTEGERVal (FirstValue_arc);
        double  _Old_Value      = __GetREALVal (OldValue_arc);
        double  _New_Value;

        if (QueueProbeTableCompute
                       (QEntry, _TableIndex, _Address, &_New_Value) > 0 &&
                (_Duplicate == 0 ||
                 _New_Value != _Old_Value ||
                 _First_Value == 0))
          {
            __PutINTEGERVal (ProbeOutput, _New_Value);
            __PutINTEGERVal (FirstValue_arc, 1);
            __PutINTEGERVal (OldValue_arc, _New_Value);
          }

      }

    __FreeArc (QueueNumber);
  }

/* -------------------------------------------------------------------- */

```

3.3. Common

3.3.1. Data Structures

3.3.1.1. IE Primitive

This Data Structure has no content.

3.3.1.2. Msg Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	INTEGER	[0,+Inf)	0
Creation Time	REAL	(-Inf,+Inf)	0.0

3.3.1.3. Msg Application Primitive

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

3.3.1.4. Msg Application Data

This Message is intended to convey an arbitrary item of data, which is modelled by a Length.

Name	*Type*	*Subrange*	*Default Value*
Length	*INTEGER*	*[0,+Inf)*	*0*
Creation Time	*REAL*	*(-Inf,+Inf)*	*0.0*

3.3.1.5. Boolean

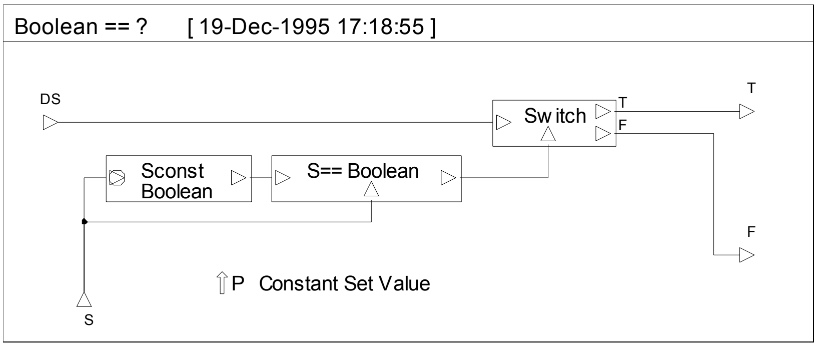
The use of a Boolean True and False is common to the extent that a data type is defined for it; as opposed to using a less consistent integer type.

Value
True
False

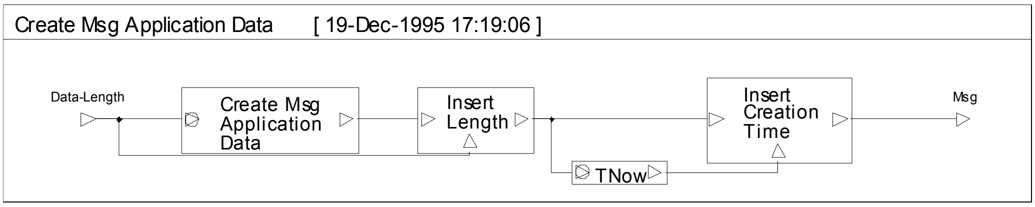
3.3.2. Modules

3.3.2.1. Boolean ==

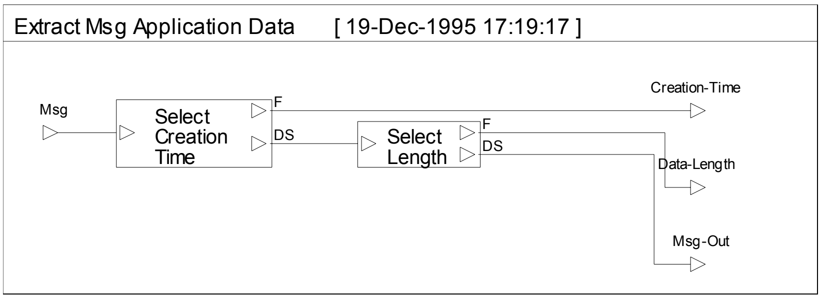
A natural operation associated with a Data Structure of this pervasiveness is that of equality evaluation. This Module attempts to evaluate such equality.



3.3.2.2. Create Msg Application Data

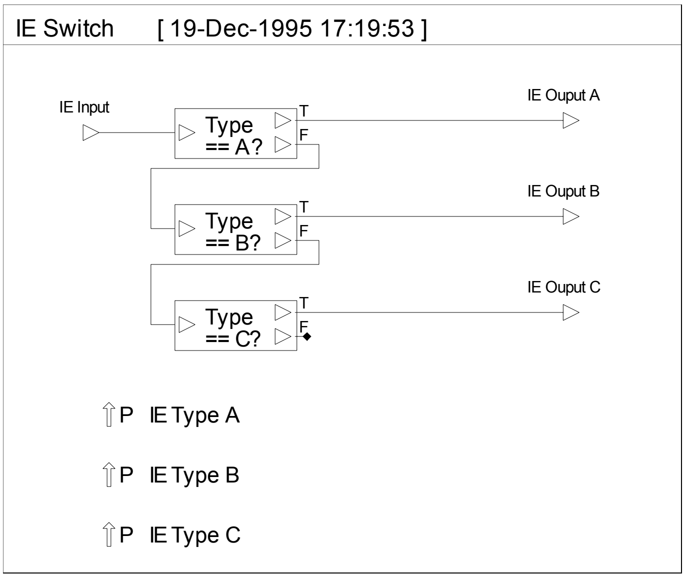

Create Msg Application Data [19-Dec-1995 17:19:06]
Data-Length
Create Msg Application Data
Insert Length
Insert Creation Time
Msg
TNow

3.3.2.3. Extract Msg Application Data



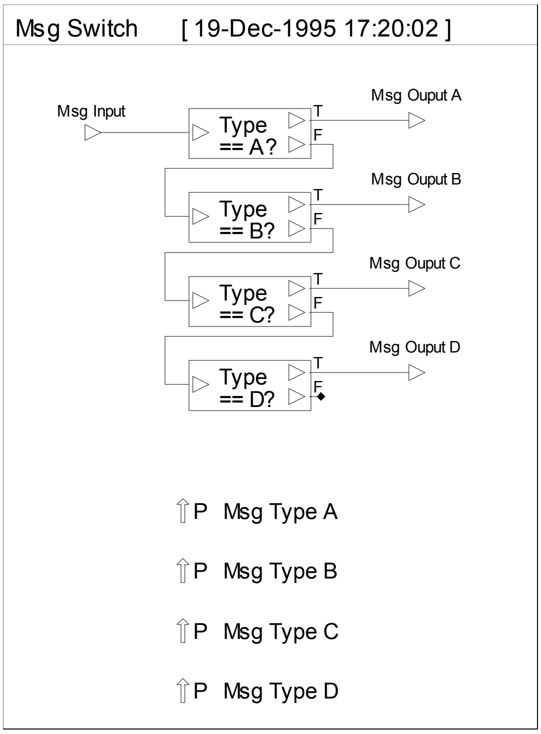
3.3.2.4. IE Switch

There are a number of places in which Information Elements are switched upon; rather than duplicate the modules required for this operation, this single Module provides the required functionality of directing an input Information Element to an appropriate output.



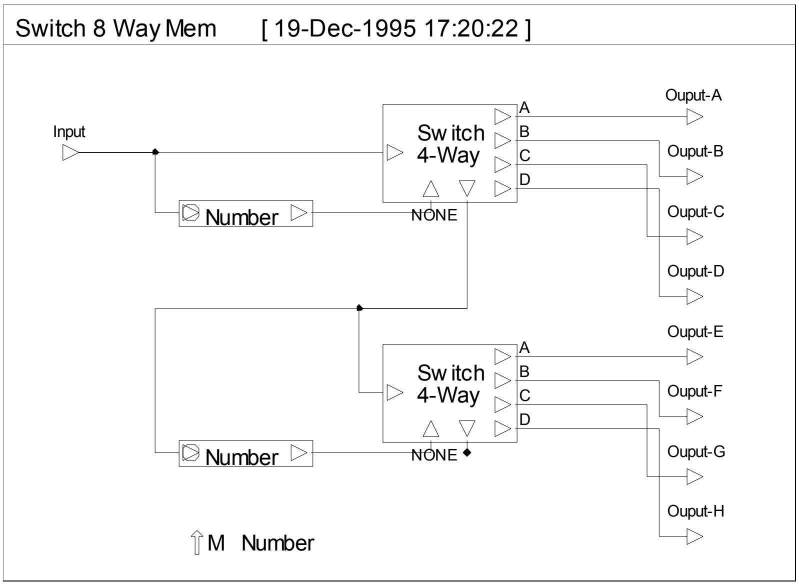
3.3.2.5. Msg Switch

There are a number of places in which Messages are switched upon; rather than duplicate the modules required for this operation, this single Module provides the required functionality of directing an input Message to an appropriate output.



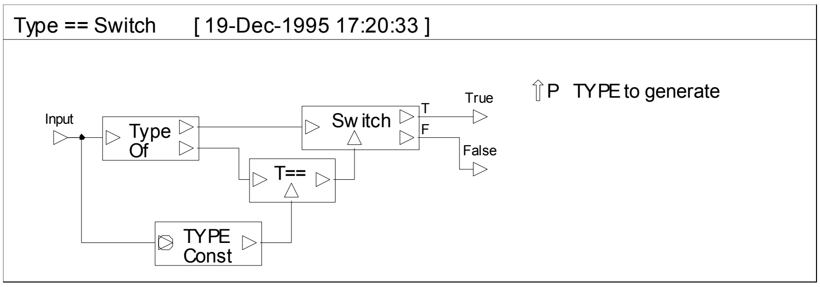
3.3.2.6. Switch 8-Way Mem

There are a couple of places where an 8 way switch is needed. A switch of this magnitude is not in the BONeS library; therefore it has been created.



3.3.2.7. Type == Switch

Switching on types is a common enough operation that it is implemented as a single module.



Type == Switch [19-Dec-1995 17:20:33]

⇧P TYPE to generate

APPENDIX 3. THESIS 1 REPORT