IPX SPOOFING PROJECT

Matthew Gream 23rd
January 1995

Introduction

This document refers to the modifications carried out to the LANPACK software in order to provide IPX spoofing services. It details what was carried out, and why it was carried out. The "Overview" section explains the operation of LANPACK and the particular problem with it's logical solution. The "Solution Implementation" section describes the current solution, not taking into account intermediate "solutions" that did not work. The following section "Problems, Limitations and Notes" does go into deal about existing problems, limitations in, and miscellaneous information about, what was finally achieved.

Overview

The LANPACK software operates on a standard PC fitted with the Jtec J2050. It's purpose is to act as an interface between an application running on the client PC and the 12050. To the client application, the software looks, and acts, like a standard packet driver. The application hands down Ethernet frames, in an ODI fragmented format, and LANP ACK shuttles these to the 12050 for transport across the link. LANP ACK also accepts Ethernet frames from the J2050 (which originated from the remote LAN the link is connected to) and passes these up to the application through the packet driver interface.

LANPACK also takes care of establishing and tearing down calls. In doing this, it can be set to connect to a particular ISDN peer "on demand", that is, when it receives a packet destined for that particular peer. It can equally operate with a "permanent" link in place. It is with the "on demand" configuration that problems occur.

If Novell is being used across the link, there are a few problems that cause unnecessary LAN calls to be made. The first problem, the trivial one, manifests itself by way of Novell's "serialisation" packets. These are periodically broadcast in order to enforce Novell's licensing agreements. They serve no other purpose, and would trigger call establishment when broadcast; fortunately they are infrequent. The second problem, which is more of a real concern, involves Novell's Service Requests (using the Netware Core Protocols, or NCP).

A Novell client can have network directories. These are directories that are not physically attached to the client, but are to the server. During the lifetime of a client<->server association (through all phases, whether a user is logged in or not), these network directories may be created, deleted or have their path names changed. The client maintains a set of "directory handles" that it associates with each particular server it is connected to. These "directory handles" map to both drive letters on the local machine, eg. A: ... Z:, and to drive letters and paths on the remote machine (eg. ADMIN\SYS:LOGIN). The server also maintains a similar map.

The client doesn't cache the drive and path names (on a server) associated with a directory handle, therefore whenever it needs to determine what drive and path a handle maps to, it must ask the particular server to provide the information. This also means that when the client wants to change a path name, it must ask the corresponding server to do so. It can also add or delete entire network drive (ie. directory handle) references it has with a server. The requests to provide the current directory path for a directory handle are called "Get Directory Path" (GDP) requests. Even with a standard DOS prompt sitting "within" a network drive, whenever return is hit and a new prompt (but looking the same) is displayed, a GDP has been executed.

A problem ...

The problem that occurs is that when a user is not using network resources but still logged into a server and for any reason a GDP is required, it will cause, for an "on demand" link, a call to be made. A prime example of this is when an application on a local drive is being used, and a fIle requestor box is

presented. The file requestor box contains network directories that can be selected *if* desired. When the file requestor was being constructed, it required a list *of* all drives on the client, and hence a series *of* GDP requests were executed. This requires a call to be made, even though the user hasn't selected (yet) to use network resources. Effectively this means that the "on demand" operation, at least with Novell, is severely limited

Corresponding to the GDP requests are "End *of* Job" (EOJ) requests that using follow each and every GDP. They are also sent when, for example, a user changes from a network drive to a local drive. In the same way as GDP's, these cause unnecessary calls to be made.

... and it's solution.

To these problems, there are solutions. In the case *of* serialisation packets, they can be "dropped" and not passed across the link. There is no harm at all in doing this.

In the case *of* the GDP requests, we can maintain a local database (cache) *of* mapping's between directory handles and path names for each particular client<->server association. We would need to trap these GDP requests, and related requests that add or delete mapping's altogether, to update and keep the database in sync. Then, when the link is down, we would "spoof' a reply to the request using information from our cache, preventing unnecessary calls from being made. *Of* course, *if* the database didn't contain the information, then a call would need to be made and the reply from the server would need to be captured to update the information.

By the same token, we can also catch EOJ requests and spoof positive confirmations back to the client. What we would need to do when the link is brought "up" is to forward these end *of* job requests to the server so that it does finally receive them.

Solution Implementation

The framework and interface with LANPACK

The first requirement is to pick out packets that are *of* interest to us. Any packets that *aren't of* interest are allowed to continue on their natural path; just as *if* no modifications to LANPACK were made. There are three interfaces with LANPACK, ignoring several other LANPACK modifications for the time being.

1. In LANPACK.C/ProcessTData:Ind() incoming fragments *of* frames from the 12050 are reassembled. When an entire Ethernet frame has been reconstructed, the application is notified by way *of* a software callback that the frame is ready to be retrieved. Before notifying the application, the frame is passed through IPX_MAIN.C/ipxs-pkt_from_link ( ) which returns an indication *of* whether the frame should be dropped or whether it should be sent. In the former case, the notify is suppressed and hence the application doesn't pick it up. It will be "overwritten" by a new packet from the link.

2. LANISR.C/TransmitPacket () is the function that accepts ODI format frames from the application and sends them off to the J2050. It contains necessary logic to determine the state *of* the link and whether or not a call needs to be made *to* establish the link. Before the frame reaches any *of* this logic, it is passed through IPX_MAIN.C/ipxs-pkt_to_link() which also returns an indication *of* whether the frame should be dropped or sent onwards. If it is to be dropped, then the function returns an immediate success.

3. Initialisation *of* the entire IPX spoofing modifications is carried out from within LANPACK.C/LANPacket_Task(} by a call to IPX_MAIN.C/ipxs_init ().

The necessary external information for LANP ACK is carried in IPXSPOOF. H. All new functions for the IPX spoofing module are contained within the files with an IPX prefix, except for an extra *two* functions and other slight modifications *to* the original LANP ACK source files. This will be described subsequently.

Logical model of the IPXSPOOF software

The following is a "basic" overview of how the IPX software is organised and structured:

Frames to the link

These are first examined to see whether or not they are frames we are interested in. This means only looking at Ethernet frames that contain IPX packets; all others are passed on. The only IPX packets we are interested in are 'Serialization' packets, which are dropped, and those containing NCP requests. Only the following NCP requests are examined:

Deallocate Directory Handle -- The mapping, if it exists in the database, for this client<->server association with this directory handle to a path name is deleted from the database. This is done because the mapping will not longer be valid after the directory handle has been deallocated. The original request is allowed to continue on to the server, establishing a call if required.

Set Directory Handle -- This too is deleted, as with "Deallocate Directory Handle". This is done because the request is to change the contents of what is in the database (ie. the request sends a new relative path name for a particular directory handle). Our strategy is to not attempt to make the modifications ourselves, but clear the entry and allow a subsequent "Get Directory Path" to refill the database. This request is also allowed to continue on to the server.

Logout -- When this occurs, all entries in the database for this particular client<->server association become invalid. The request is allowed to continue on to the server.

End of Job -- If the link is currently connected, then this request is allowed to continue on to the server without any further processing. However, if the link is not connected then a "spoofed" reply is created and returned to the application. Thus, the application "thinks" the server has accepted and responded to the request. What happens next is that the software records the fact that a request on a particular client<->server has been spoofed, and takes note of the sequence number of the request. The original request is not allowed to continue on to the server.

Get Directory Path -- If the link is currently connected, then the first thing that occurs is that any entry that exists in the database for this client<->server association and directory handle is deleted. A "hook" is put in that will capture the NCP reply for this request as it comes back from the server, and the request is allowed to continue on to the server. The reply, when trapped, will contain the current path name information, and will be used to update the database. This ensures that the database always contains the newest information; note that a negative reply won't be added to the database, if it occurs. If the link is not connected and the database doesn't contain a corresponding entry, then the request is also allowed to continue on to the server, again with the same "hook" mechanism put into place. Finally, if the link is not connected and the database does contain an entry, a spoofed reply is generated and returned back to the application. The fact that this sequence number was not sent is recorded, as explained in the case of an 'End of Job'. The request is not allowed to continue on to the server, in that case.

Frames from the link

The only frames of interest that come from the link are NCP replies. In particular, the replies to a "Get Directory Path" request are usually required to be used to fill in an entry in the database. The software maintains a list of replies that it needs to examine, and each NCP reply is examined to determine whether or not it matches an entry in the list. In the case of a "Get Directory Path"' reply, the database entry is only updated if the reply contains a positive indication of the request (a failure could occur if an incorrect directory handle was specified). Note that the reply contains the actual path name for the directory handle that was originally sent in the request. These replies are allowed to continue on to the client.

There is an additional detail not yet covered, that is the what occurs when the link becomes active and the LANPACK software has a record of the sequence numbers of spoofed request/replies for a particular client<->server association. Basically, the problem is that each NCP request/reply has a sequence number that increases linearly modulo 256. The server will silently ignore any incoming requests that are out of order (ie. not equal to the next sequence number that is expected), which may occur if a request is lost

and a second request arrives. The client's strategy is to time out and resend the outstanding requests. This is a valid mode of operation, as commands do necessarily rely on the successful operation of previous commands.

When client to server requests are spoofed, the client receives a reply for that request, and hence a confirmation that the sequence number associated with that request was accepted, but the server never "sees" that sequence number. When the link becomes active, and client requests are passed to the server with or without action by this software, they will contain sequence numbers higher than that the server expects. Again, this is because the server has never seen those intermediate sequence numbers, we "faked" them. The result is the client goes through a series of timeouts and the only means of recovery is to abort the connection. This is fairly nasty.

There were four strategies that could have rectified this situation, they are documented in an associated file "PROBLEMS.DOC". Only one strategy was anywhere near practical, and what it involved was keeping a note of the missing sequence numbers while the link is not connected --- as noted in the operation of the software above. When the link is connected, it is required to transmit a number of "dummy" requests to the server so that it will "see" the missing sequence numbers. This must occur before the client in the association times out it's sending of the next request (ie. the one that has a sequence number starting just after the gap to be filled in). This is where the problem lies, because with a possible maximum of 255 "dummy" packets to send, meeting this timeout bound isn't likely. In addition, should any of these be lost, it would not be possible to recover.

These "dummy" requests are sent when the first request destined on a particular client<->server is detected by the software. Because if the out of sequence mechanism. the software cannot send the next "dummy" request until it has received a reply on the current request. Hence, to pace itself, it traps the reply for the request and then transmits the next request. This continues until it has "filled in the gap".

That completes a, largely but not strictly, overview of how the software operates without explaining particular implementation oddities. For a more complete look at the software, each module will be examined in turn.

IPX spoofing software module breakdown

IPX_MAIN. C

This module provides functions that are called from within the LANP ACK software, namely:

ipxs_pkt_to_link() -- The frames passed into here are not "contiguous", they are in the fragmented (001) format. A contiguous frame is extracted but only up to a certain length. The reason for this is that frames we are interested in will never exceed a certain length, hence there is no reason for copying, or even considering for that matter, frames that are "too long". Following this, the locations of the sub frames are determined (Ethernet, IPX, NCP) and if the NCP packet is a request it is handed off to ipxs_proc_ncp_request () to be further examined. If the IPX packet is a serialisation packet, it is dropped. Anything else is allowed to pass on without alteration. Note that there are hooks in there for setup_sig_complete () if the action is to drop a frame This is a function that causes a "packet transmitted successfully" interrupt to be made to the application. It is required because the application must be notified of the transmit status of every frame it sends. When a packet is dropped, we still want the application to think it was sent successfully, hence the "spoofing" of a transmit complete.

ipxs_pkt_from_link() -- Frames coming from the link destined for the application are passed in here. As with the previous function, the location of the sub frames is determined and only those that are NCP replies are handed off to ipxs_proc_ncp_reply () for further action. Everything else is allowed to pass on without alteration.

ipxs_init ( ) -- Performs initialisation of anything within the IPX spoofing software that requires so.

IPX_PROC.C

This module provides the upper level functions that examine and act upon the contents *of* NCP requests and replies. There are two functions *of* importance:

ipxs_proc_ncp_request () _ Frames containing NCP requests are passed into here. The type *of* request must be a "Set Directory Handle", "Deallocate Directory Handle", "Get Directory Path", "Logout" or "End *of* Job" or the function will exit allowing the frame to travel onwards unaltered. The source and destination addresses from the IPX packet are used as "keys" in the internal databases. In addition, the database *of* directory paths uses the directory handle extracted from the NCP request as another key, and the database *of* confirmations uses the uses the sequence number and connection number from the NCP request as it's other key. These are extracted before further processing.

If the request is a "Set Directory Handle" or "Deallocate Directory Handle", then the corresponding entry in the directory path database is removed *if* it exists. Note that *if* a confirmation has not yet been received for this entry, then it is also cleared out from the confirmation database. The request is allowed to continue onwards.

If the request is a "Get Directory Path", then the corresponding entry is looked up in the directory database, *if* the link is currently IDLE. If the entry exists, then a spoofed reply is generated and sent back to the application, and the previously mentioned "unsent sequence numbers" database is updated. The request is then dropped. However, *if* the link isn't IDLE *or* an entry wasn't found in the database, then a blank entry is allocated in the directory path database, and a hook is put into the confirmation database to trap the returned reply. This request is then allowed to continue onwards.

If the request is an "End *of* Job" *and* the link is IDLE then a spoofed reply is generated and sent back to the application. The "unsent sequence numbers" database is updated to indicate that another spoofed packet has been sent, and this request is then dropped. IF the link isn't IDLE then the request is passed on untouched.

If the request is a "Logout", then all matching entries in the directory path database are removed corresponding to the address pair for the association. This erases all directory handle mapping's for that particular session, noting also that any outstanding confirmations connected to these entries are also removed.

ipxs_proc_ncp_reply ( ) -- Here the incoming frames have their respective key information extracted, as just noted with respect to the processing *of* requests. This key information is used to locate an entry in the confirmation database, and *if* none exists then the function returns allowing the frame to continue on without modification. If a match is located, then action is performed according to the type *of* confirmation hook that was constructed. Note that the confirmation entry is removed from the database before anything else is done, because now it is obsolete.

If the hook is *of* type "CF _E_ADD", then the reply is a response to a "Get Directory Path". If the reply contains a positive confirmation, then the new path can be copied into it's respective position in the directory path database. If the confirmation is negative, then the database entry is removed. In both cases, the frame is then allowed to continue on without modification.

IPX_REPL.C

There are two main routines in this module which provides for the construction *of* spoofed replies. The first, reply _gdp () constructs a reply to a "Get Directory Path" request and reply_eoj () constructs a reply to an "End *of* Job" request. Both *of* these then forward (or spoof) the reply back to the application.

There are a few caveats. Firstly, IPX checksums are not computed. For the moment this is OK as it is possible, and usual, for IPX checksums to be turned off. But it is definitely possible for this product to be deployed in an environment that does use checksums. Something should be done about this at a later date. Secondly, the FCS field *of* the Ethernet frame is not computed, but this

is not a problem because the LANP ACK software strips this off before handing the frame up *to* the application.

The way frames are "sent back" *to* the application *is* also worth mentioning. What occurs *is* that the main LANP ACK database contains an entry to hold a spoofed frame, and this *is* where it *is* constructed. The module indicates that a spoofed frame *is* ready *to* be sent by sending a T_DATA_IND back *to* LANPACK, ie. itself. LANPACK will then, as part *of* it's normal processing, pick out this message and proceed *to* pass the spoofed frame up to the application. The reason the frame isn't directly sent *to* the application *is* because at the time it has *to* be sent, the software *is* in a software interrupt from the application. It would be unwise *to* directly send it, which involves making a call back *to* the application, while already in a call from the application.

In addition, there *is* a function setup_sig_complete () that *is* responsible for faking a T_DATA_CON. The J2050 indicates the fact that a frame has been successfully transmitted by sending a T_DATA_CON *to* LANPACK, which then "calls" the application *to* inform it *of* a successful transmit When dropping frames, because *of* an IPX spoofing requirement, it *is* still required that the application be informed that a successful transmit occurred. Therefore, this function provides that It *is* called from XPX_MAXN. C whenever a frame *is* going to be dropped.

IPX_GDP.C

This module acts as the means by which the IPX spoofing software keeps track *of* missing sequence numbers corresponding *to* spoofed replies. Any time a reply *is* spoofed, the update function *is* called. It takes care *of* keeping track *of* a table indicating the range *of* sequence numbers that have been "missed" on a particular association because *of* spoofing. The function also takes a copy *of* the first frame it *is* passed and modifies it *to* act as a later "dummy" request

When the link becomes active and a request *is* seen on an association that had replies spoofed during the link idle period, then the strategy *is to* send a number *of* "dummy" requests *to* make up for the sequence number space that the server "never saw". It sends the fIrst "dummy" request and then puts in a hook *to* catch the returned reply from the server. When it gets this reply it sends the next "dummy" and this process continues until all the "missing" sequence numbers have been transmitted.

There *is* currently a problem with this strategy, this *is* detailed further in the problems and limitations section (subsequently).

IPX_DB.C

This module implements the directory path database and provides all the necessary access functions. There *is* also a corresponding header file with element definitions and prototypes. The database has *two* keys, the first being a "directory handle" and the second being a part *of* addresses for a particular association (ie. the IPX net and node addresses for *both* source and destination). Entries can be added *or* removed from the table *of* active entries, but note that there *is* a fixed maximum size for the table; this *is* because all memory must be preallocated. The active table *is* implemented using hashing for fast(er) access. Note also that timestamps are used on each added entry *so* that "old" entries can be easily located and removed when the table *is* full. Each entry has space *to* contain a directory path, and a pointer *to* an entry in the confirmation database.

IPX_CF.C

This module, the confirmation database, *is* very similar to the directory path database; except that in this instance what *is* being stored are hooks *to* match and act upon NCP replies. The keys for the database are a sequence number and connection number along with the address pair for a particular association. Each entry stores an identifier that indicates what action *is to* be taken upon reception *of* the confirmation and includes a pointer to an entry in the directory path database, should there be one *to* match up *to.*

IPX_DEBU. C

Contains debug reporting routines that use the Report interface provided within JEXEC. Not very important in the scheme of things.

IPXSPOOF.B

This header file acts as the external interface to the IPX spoofing software. It contains the necessary prototypes and definitions needed by the rest of the LANP ACK software.

IPX_PRIV.B

This header fIle contains prototypes, definitions and details specific only to the IPX spoofing modules themselves, it does not have any external scope.

Other modifications to LANPACK software

There were a few minor modifications to the LANP ACK software itself to support the IPX spoofing software. First, several additions were made to the database so it could hold a spoofed frame, ready to be sent, the length of the frame and an indication that the frame is currently being "processed". In addition, when sending "dummy" requests after a link has become active, the J2050 will providing transmit complete indications (T_DATA_CON) that should not be passed on to the application. Hence, a counter has been instrumented that keeps track of the number ofT_DATA_CON's that should be ignored.

Modifications were made the following:

LANCFG.C/:InitialiseDatabase () -- Initialise the new database entries to default values.
LAN:ISR.C/TransmitPacket () -- This is where ipxs_pkt_to_link() is called from and its return indicates whether or not the frame should continue on to be transmitted, or should be dropped.
LAN:ISR.C/PollPacketResult () -- This is called when the application wants to receive a frame it knows is waiting; therefore modifications are in place to provide the spoofed frame if it is waiting.
LANRTN.C/ResetCallDatabase() -- Additions to support the suppression of T_DATA_CONs.
LANPACK.C/LANPacket_Task() - Calls ipxs_init () to start up the IPX spoofing additions. Incoming T_DATA_IND's are checked to see whether the message was generated locally, and hence is a spoof. Incoming T _DATA_CON's are not processed if any suppression of them is required.
LANPACK.C/PrintStatus () -- Additions to show values of new variables in the database.
LANPACK.C/ProcessTData:Ind() - Calls ipxs_pkt_from_link() and indicates, by return, whether or not the frame should be dropped or allowed to progress to the application.
LANPACK.C/Process:IpxSpoof () -- New function to hand up spoofed frame to application.
LANPACK.C/ProcessSpoofCon() - New function to signal a transmit complete (to application) regardless of current link state.
LANPACK.C/ProcessOpenFailure() -- Additions to suppress indication of successful transmit complete if suppression of T_DA T A_CONs is still required.

Problems, Limitations and Notes

There are a number of items that are best presented by describing them in point form:

1. The keys for the databases must include both the source and destination IPX net and node addresses as keys, this is because it is possible to have multiple clients talking to multiple servers. It looks as though, and it's a safe assumption that, directory handles are unique only within the domain of a particular client/server; especially as it is the server that allocates the handle when the server has no

idea of other servers the client is communicating with. Note that port numbers do not bear any significance, at least in the context we are concerned with.

2. As it stands, there is (what I would consider) significant overhead in terms of having to match addresses in the databases. An optimal way to rewrite this would be to tree the database entries down from a small address table, this would minimise the number of address comparisons. There are a few other ways to achieve the same goal though. This is a fairly minor concern, and only warrants worrying about if it is found that frames are being lost due to the overhead in LANPACK.

3. The "dummy" request mechanism doesn't work at the moment What happens is that when the link is brought up, the first dummy request is successfully sent and acknowledged, but the second does not get an acknowledgment. This occurs whatever type of dummy request is used, and the problem has not been located. More than likely it is a problem with the LANPACK/IPX software as there is no logical explanation as to why the server should reject a request. The current requests being sent are NDS "ping" requests; these should always have replies forthcoming. What has been tested is setting up a scenario where only one sequence number is "missing", therefore only one dummy request need be sent. This worked.

   Note that even if this problem was solvable there is still the question of the excessive time required to send all the dummy requests. In addition, support should be added for the possible situation where a dummy request is "lost"; this would add significant overhead to the software, but is a definite requirement if this strategy is to be pursued.

4. It is important to note that frames going to the link are in the 001 format where they are fragmented, so for examination the fragments need to be aggregated. They must sent to the J2050 in a contiguous form. however.

5. One major problem is that of logging; too much logging ends up causing significant problems. The JDISK task ends up spending considerable time writing out to disk and as a consequence, some frames are lost and never seen by LANP ACK. This causes error messages at the client, requiring manual intervention. When developing, this must be taken into consideration -- though clearly these debug messages are taken out in the "final" version and won't cause a problem then.