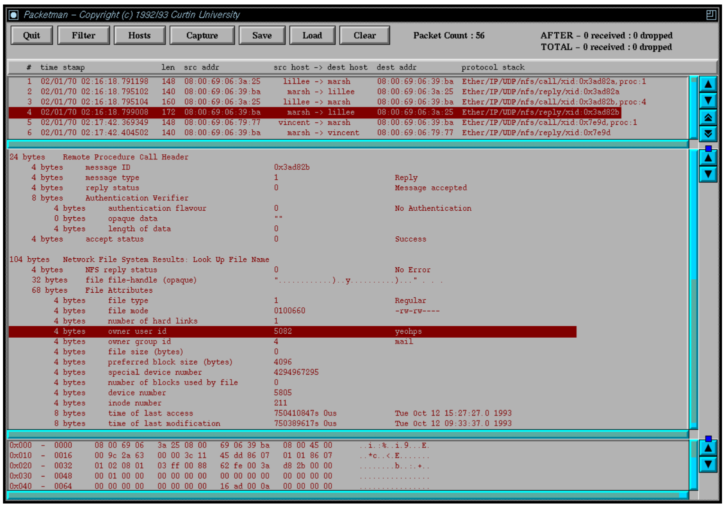


Packetman v1.1
26 November, 1993

Name: Gregory Barron
e-mail: pug@cs.curtin.edu.au
Student #: 892248k
Unit: Computing Project 394

Supervisor: Craig Farrell.
e-mail: craig@cs.curtin.edu.au

Table of Contents

List of Figures vi
List of Tables viii
Abstract x
Preface xii
1.0 Introduction 1
2.0 Background 1
2.1 Protocol Monitoring 1
2.2 Protocol Analysis Tools 2
2.2.1 Analyzer 2
2.2.2 Network General Sniffer 2
2.2.3 Nfswatch 3
2.2.4 Tcpdump 5
2.2.5 Tcpview 6
3.0 Packetman v1.0 7
3.1 Overview 7
3.2 Mechanism 10
3.2.1 Capturing Packets 10
3.2.1.1 Network Interfaces 10
3.2.1.1.1 SunOS: Network Interface Tap (NIT) 10
3.2.1.1.2 Ultrix: Packetfiter 11
3.2.1.2 Storing Packets 11
3.2.1.2.1 Filtering 11
3.2.1.2.2 Copying 12
3.2.2 Displaying Packets 12
3.2.2.1 List Widgets 12
3.2.2.2 Packet Decode List 13
3.2.2.3 Packet Dump List 13
3.2.3 Decoding Packets 13
3.2.3.1 Overview 13
3.2.3.2 Decoding a Telnet Packet 13
3.2.3.2.1 Ethernet Header 14
3.2.3.2.2 IP Header 14
3.2.3.2.3 TCP Header 14
3.2.3.2.4 Telnet Data 15
3.3 Evaluation 15
3.3.1 Strengths 15
3.3.2 Deficiencies 15
4.0 Facilities Used 16
4.1 Workstations 16
4.2 Revision Control System (RCS) 17
5.0 Aims 17
6.0 Changes 18
6.1 X11 Interface 18

6.1.1 Processing Events 18
6.1.2 Repositioning Widgets..19
6.1.2.1 Arrows 19
6.1.2.2 Counter 20
6.1.2.3 Statistics Pane..21
6.1.3 Dialogs 21
6.1.3.1 Capture Dialog..21
6.1.3.2 File Dialogs 22
6.1.3.3 Host Dialog..22
6.2 Arguments 23
6.3 Sniffer Files 23
6.4 Protocols..24
6.4.1 Internet Control Message Protocol (ICMP)..24
6.4.1.1 Overview 24
6.4.1.2 Message Types 24
6.4.1.2.1 Destination Unreachable Message..24
6.4.1.2.2 Time Exceeded Message..26
6.4.1.2.3 Parameter Problem Message..26
6.4.1.2.4 Source Quench Message 27
6.4.1.2.5 Redirect Message 27
6.4.1.2.6 Echo or Echo Reply Message 28
6.4.1.2.7 Timestamp and Timestamp Reply Message 28
6.4.1.2.8 Information Request or Reply..29
6.4.1.2.9 Address Mask Request and Address Mask Reply 29
6.4.1.3 Implementation..29
6.4.2 Remote Procedure Calls (RPC) 31
6.4.2.1 Overview 31
6.4.2.2 eXternal Data Representation (XDR)..31
6.4.2.3 Rpcgen..32
6.4.2.4 Message Structure 33
6.4.2.4.1 Call Messages 33
6.4.2.4.2 Reply Messages..33
6.4.2.4.3 Authentication..34
6.4.2.4.3.1 Null Authentication (AUTH_NULL)..34
6.4.2.4.3.2 Unix Authentication (AUTH_UNIX)..34
6.4.2.4.3.3 Short Hand Unix Authentication (AUTH_SHORT) 35
6.4.2.4.3.4 DES Authentication (AUTH_DES)..35
6.4.2.5 Implementation..35
6.4.2.5.1 XDR 35
6.4.2.5.2 Matching Call/Reply Messages 35
6.4.2.5.2.1 Arrays 36
6.4.2.5.2.2 Height Balanced (AVL) Trees..36
6.4.2.5.2.3 Hashtables..36
6.4.2.5.3 Decoding 37
6.4.2.5.3.1 Overview 37
6.4.2.5.3.2 Call Messages 37

6.4.2.5.3.3 Reply Messages 39
6.4.2.5.3.4 Half Decoding...43
6.4.3 Network File System (NFS)..44
6.4.3.1 Overview 44
6.4.3.2 NFS Procedures 45
6.4.3.2.1 Null Procedure 45
6.4.3.2.2 Get File Attributes..45
6.4.3.2.3 Set File Attributes 46
6.4.3.2.4 Get Filesystem Root..46
6.4.3.2.5 Look Up File Name..46
6.4.3.2.6 Read From Symbolic Link 47
6.4.3.2.7 Read From File..47
6.4.3.2.8 Write To Cache 48
6.4.3.2.9 Write To File..48
6.4.3.2.10 Create File 49
6.4.3.2.11 Remove File 49
6.4.3.2.12 Rename File 49
6.4.3.2.13 Create Link To File 50
6.4.3.2.14 Create Symbolic Link 50
6.4.3.2.15 Create Directory..51
6.4.3.2.16 Remove Directory 51
6.4.3.2.17 Read From Directory 51
6.4.3.2.18 Get Filesystem Attributes..52
6.4.3.3 Implementation..53
6.4.3.3.1 Data Structures..53
6.4.3.3.2 Decoding RPC Headers 54
6.4.3.3.3 Decoding RPC Programs 55
6.4.3.3.4 Name Resolution..56
6.4.4 NFS Mount..57
6.4.4.1 Overview 57
6.4.4.2 Mount Procedures..58
6.4.4.2.1 Add Mount Entry 58
6.4.4.2.2 Remove Mount Entry..58
6.4.4.2.3 Remove All Mount Entries 58
6.4.4.2.4 Return Export List..58
6.4.5 Portmapper (SunRPC) 58
6.4.5.1 Overview 58
6.4.5.2 Portmapper Procedures..59
6.4.5.2.1 Register Program..59
6.4.5.2.2 De-register Program..59
6.4.5.2.3 Get Program Mapping..59
6.4.5.2.4 Get Program Mappings 59
6.4.5.2.5 Call Remote Procedure 60
7.0 Future Directions 60
7.1 BSD Packet Filter..60
7.2 File Selection Widget 60

7.3 Filtering 61
7.4 Information Widget 62
7.5 Options Widget....62
7.6 Porting 62
7.7 Protocols....62
8.0 Conclusion 62
9.0 References....65
Appendix A — Glossary....67
Appendix B — Manual Page 71
Appendix C — Noctools Entry....75

List of Figures

Figure 1: Sample *nfswatch* output. 4
Figure 2: Sample *tcpdump* output. 5
Figure 3: *tcpview* decoding an NFS packet. 6
Figure 4: The *packetman* display. 8
Figure 5: The main *packetman* controls. 9
Figure 6: Packet trace pane. 9
Figure 7: Packet decode pane. 9
Figure 8: Packet dump pane. 10
Figure 9: Statistics pane. 10
Figure 10: Internal buffer definitions. 12
Figure 11: Telnet ethernet frame. 14
Figure 12: Telnet IP header. 14
Figure 13: Telnet TCP header. 15
Figure 14: Telnet data. 15
Figure 15: Host filtering dialog. 22
Figure 16: *Packetman* v1.1 display. 23
Figure 17: Destination Unreachable message format. 24
Figure 18: Destination Unreachable/Port Unreachable message. 25
Figure 20: Time Exceeded/TTL Exceeded Message. 26
Figure 19: Time Exceeded message format. 26
Figure 21: Parameter Problem message format. 26
Figure 23: Source Quench message. 27
Figure 25: Redirect Message/Datagram For Host message. 27
Figure 22: Source Quench message format. 27
Figure 24: Redirect message format. 27
Figure 27: Echo/Echo reply messages. 28
Figure 26: Echo/Echo Reply message format. 28
Figure 28: Timestamp/Timestamp Reply message format. 28
Figure 29: Timestamp/Timestamp Reply message. 29
Figure 30: Information Request/Reply message format. 29
Figure 31: Address Mask Request/Reply message format. 29
Figure 32: RPC operation (Corbin, 1991). 31
Figure 33: NFS settable attributes definition. 32
Figure 34: NFS settable attributes XDR filter. 33
Figure 35: RPC call message format. 33
Figure 36: RPC reply message format. 34
Figure 37: RPC call header. 39
Figure 38: RPC reply header. 43
Figure 39: RPC protocol stack. 44
Figure 40: Get File Attribute arguments. 45
Figure 41: Get File Attribute results. 45
Figure 42: Set File Attributes arguments. 46
Figure 43: Set File Attributes results. 46

Figure 44: Look Up File Name arguments. 46
Figure 45: Look Up File Name results. 47
Figure 46: Read From Symbolic Link arguments. 47
Figure 47: Read From Symbolic Link results...47
Figure 48: Read From File arguments. 47
Figure 49: Read From File results. 48
Figure 50: Write To File arguments. 48
Figure 51: Write To File results...48
Figure 52: Create File arguments...49
Figure 53: Create File results...49
Figure 54: Remove File arguments...49
Figure 55: Remove File results...49
Figure 56: Rename File arguments...49
Figure 57: Rename File results. 50
Figure 58: Create Link To File arguments...50
Figure 59: Create Link To File results...50
Figure 60: Create Symbolic Link arguments...50
Figure 61: Create Symbolic Link results. 50
Figure 62: Create Directory arguments. 51
Figure 63: Create Directory results..51
Figure 64: Remove Directory arguments..51
Figure 65: Remove Directory results..51
Figure 66: Read From Directory arguments. 51
Figure 67: Read From Directory results. 52
Figure 68: Get Filesystem Attributes arguments. 52
Figure 69: Get Filesystem Attributes results. 52
Figure 70: Add Mount Entry arguments...58
Figure 71: Add Mount Entry results. 58
Figure 72: Remove Mount Entry arguments. 58
Figure 73: Get Program Mapping arguments. 59
Figure 74: Get Program Mapping results. 59
Figure 75: Call Remote Procedure arguments...60
Figure 76: Call Remote Procedure results. 60
Figure 77: File Selection Widget (Frame Technology Corp., 1992) 61
Figure 78: Filter Selection Widget...61

List of Tables

Table 1: Compilation time for various architectures. 17
Table 2: Counter increment values. 21

Abstract

The increase in the number, size and complexity of networks and network protocols has increased the need for simple and easy to use network management tools. There are many types of such tools, one of which is the protocol analyser.

This report describes the extension of one such analyser, *packetman*, which was developed during 1992 at Curtin University as part of the *netman* project. This report describes the changes made to *packetman* to make it a more useful tool, as well as the basics behind the operation of computer protocols — specifically those of RPC and NFS. Areas of future development are described in the conclusion.

Preface

There are many people who contributed to this project in many ways, all of which deserve much thanks. Specifically, I'd like to thank Craig Farrell and Mike Schulze, for their constant guidance, ideas, solutions, encouragement and threats throughout the course of the project.

Thanks also to Dave Curry and Jeff Mogul for *nfswatch*, which was studied to find out just how RPC worked — I promise I used none of your code! Also, thanks to Steve McCanne, Van Jacobson and Craig Leres for *tcpdump*, which was also studied.

Peter Staubach from Sun helped out by sending the latest version of rpcgen (it was a life-saver) as well as the nfs.x file for version 3 of NFS — I promise I'll get around to version 3 soon. Thanks also to Martin Hunt for the sniffer code which he allowed me to use. Thanks to the many people on the net who helped out with suggestions, bug reports and solutions when things got too hard. I hope you like the result.

Lastly and most importantly I'd like to thank George Benko, who gave me an excellent foundation from which to work. Without his effort none of this would have been possible.

1.0 Introduction

The proliferation of computers in recent times has increased the size, complexity and number of computer networks — all of which need to be constantly maintained.

To this end, many tools have been developed to help accomplish this task. These tools come in many forms, from graphical displays of network traffic (*etherman*) to text dumps of packet headers (*tcpdump*).

Another such tool is a protocol analyser, which can be used to decompose packets captured from the network. This report describes *packetman*, a protocol analyser developed at Curtin University during 1992, as well as the enhancements made to it during the course of this project.

It is hoped that this report will not only be useful as a reference to *packetman*, but also be used as a tutorial towards the understanding of network traffic, specifically that of Remote Procedure Calls (RPC) and the Network File System (NFS).

The remainder of this report is formatted as follows: Section 2.0 explains the background to the area as well as the concept of protocol monitoring; Section 3.0 describes *packetman* v1.0, the version of *packetman* at the beginning of this project; Section 4.0 describes the facilities used to complete the project; Section 5.0 outlines the major aims of the project; Section 6.0 describes all changes made during the course of the project; Section 7.0 outlines the ways that *Packetman* may be improved in the future; Section 8.0 concludes the report; Appendix A is a glossary of unusual terms; Appendix B contains the *packetman* manual page; Appendix C contains the *packetman* noctools entry.

Because of its technical nature, this report contains terms which may be unfamiliar to some readers. It is recommended that a reader unfamiliar with this area peruse appendix A before continuing.

2.0 Background

2.1 Protocol Monitoring

A protocol is a set of rules which two computers can use to communicate with one another. There are many types of protocols, all of which differ in their frequency, complexity and purpose. The operation of a network is built upon the implementations of various protocols.

Because of their importance to a network, it is necessary to monitor protocol traffic, particularly when sporadic problems are occurring. It is for this purpose that protocol monitors (packet analysers, protocol analysers), are used.

A protocol analyser has two main uses:

- Diagnosis: Tracing protocols can help to identify any problems which may be occurring.
- Education: Viewing the network to increase the user’s understanding of network communications. Dumps of actual packets serve as an excellent supplement to protocol descrip-

tions.

To be successful as both a diagnosis and educational tool, a protocol analyser must provide (at least) the following functions:

- It must read packets promiscuously. i.e. It must read all packets present on the network.
- It must provide filtering mechanisms such that the user can filter out (in) unwanted (wanted) protocols.
- The decomposition of packets must be formatted such that they are easily read and understood.
- It should be easy to use. A GUI can usually be used to hide much of the low level detail from the user.

2.2 Protocol Analysis Tools

There are many tools which can be used to monitor and decipher protocol information. Some of these were studied before *packetman* was implemented, in order to find a frame of reference. As an example of this, the *packetman* display was modelled after that of the Network General *Sniffer*.

Some of these tools are described in this section.

2.2.1 Analyzer

Analyzer is the protocol analyser included in Silicon Graphics' *NetVisualyzer* package. *NetVisualyzer* consists of several graphical tools that provide for real-time network monitoring and diagnosis (Silicon Graphics Inc.).

The display for *Analyzer* is segmented into three sections, similar to that of the *Sniffer*, described below. The *Analyzer*, however, offers a more intuitive interface than the *Sniffer* — it is based on the *gl* library — and is therefore easier to use.

Analyzer includes filtering capabilities, though the author is unable to comment on their complexity or compare them with those of the *Sniffer*. The filters are shared across the entire *NetVisualyzer* package and can be customised to a complex level.

Along with the entire *NetVisualyzer* package, *Analyzer* is only available for the IRIX environment as a commercial product, and is therefore not available in source form. The expense of this product has precluded a detailed analysis.

2.2.2 Network General Sniffer

The *Network General Sniffer* is a PC based protocol analyser, network monitor and traffic generator that can be used for LAN and WAN diagnostics (NGC, 1992). The *Sniffer* is a complex hardware/software product which consists of a modified ethernet card and 128k buffer encased within a portable stand-alone unit. It is the tool against which most network monitoring tools are compared.

When in protocol analyser mode, the *Sniffer's* display is segmented into three sections. The top section shows brief information about the packets that reside in the buffer. The middle section displays a detailed description of the packet selected in the top window. The bottom section displays a simple hex dump of the current packet.

The *Sniffer's* major characteristics are:

- The *Sniffer* can fully decode many well used protocols including: IBM Token-Ring, Novell Netware, DECnet, TCP/IP, X11, NFS, RPC as well as several RPC programs. In addition, all seven layers of the OSI model can be decoded. It is also possible for users to write their own decoding routines, which can then be linked with the Sniffer software and used. Such routines have been written and placed in the public domain.

- The user interface is more than the command line of some network tools (e.g. *tcpdump*), using a custom menu system to choose from its many options.

- The *Sniffer* provides powerful filtering which enables the user to capture only those packets which are of interest. The filtering mechanism allows the user to filter at different levels, including protocol content and node addresses.

- The *Sniffer* is a self contained unit, and can be plugged straight into many LANs, including ethernet, Token-ring and Apple Localtalk.

The *Sniffer* is not without its faults, however:

- Though the interface is an improvement over other tools, for the novice it leaves much to be desired. When it is considered that most users of the *Sniffer* will be systems administrators, this is not an extremely important consideration.

- The *Sniffer's* major shortcoming is it's price. When Curtin University purchased its *Sniffer* several years ago, the cost was approximately $(Aus) 50, 000 — although this has decreased in recent years due to competition (S. Maddocks, pers. comm., 1993). For many organisations its cost will be prohibitive.

2.2.3 Nfswatch

Nfswatch (Curry and Mogul, 1993) can be used to monitor traffic between an NFS server and its client or clients. *Nfswatch* listens to all traffic destined for the server, and records statistics of the packets received.

Though *nfswatch's* primary task is to record NFS statistics, it will also record statistics about other protocols it can identify. The information collected is displayed in a textual format. An example of an *nfswatch* execution appears in figure 1:

```
lillee.cs.curtin.edu.au     Tue Nov  9 16:47:25 1993     Elapsed time: 00:15:50
Interval packets:      1674 (network)        377 (to host)          0 (dropped)
```

```
Total packets:       358880 (network)      59320 (to host)        77 (dropped)
                      Monitoring packets from interface le0
                     int   pct   total                          int   pct   total
ND Read                0    0%       0 TCP Packets               47   12%    6950
ND Write               0    0%       0 UDP Packets              328   87%   52049
NFS Read               4    1%    1448 ICMP Packets               0    0%      16
NFS Write              0    0%     345 Routing Control            6    2%     192
NFS Mount              0    0%       1 Address Resolution         0    0%       6
YP/NIS/NIS+            0    0%     114 Reverse Addr Resol         0    0%       0
RPC Authorization    312   83%   48208 Ethernet/FDDI Bdcst       11    3%     443
Other RPC Packets      0    0%     207 Other Packets              2    1%     299
                                18 NFS Procedures
Procedure           int   pct   total  completed  ave.resp  var.resp  max.resp
  CREATE              0    0%      41
  GETATTR             0    0%     291
  GETROOT             0    0%       0
  LINK                0    0%       0
  LOOKUP              0    0%     442
  MKDIR               0    0%       0
  NULLPROC            4  100%     444
  READ                0    0%     155
  READDIR             0    0%      12
  READLINK            0    0%     103
  REMOVE              0    0%      16
  RENAME              0    0%       8
  RMDIR               0    0%       0
  SETATTR             0    0%      18
  STATFS              0    0%       1
  SYMLINK             0    0%       0
  WCACHE              0    0%       0
  WRITE               0    0%     262
```

Figure 1: Sample *nfswatch* output.

As can be seen, *nfswatch* groups NFS packets according to their high level activity. i.e. If the packet is a write operation, then it is recorded as an NFS write. Similarly read operations are recorded as NFS reads. When executed with the `-proc` switch, as above, *nfswatch* will also record statistics of each type of NFS packet.

The statistics for the NFS packets are only those of the call messages — reply packets are not matched with their corresponding calls. Reply messages can be matched quite easily by storing information about each call message. The message id, program number, program version and procedure number of such messages can be stored. This information can then used to identify the reply packet, by matching up the message ids. Once two packets have been matched the information stored can be cleared.

Though at first glance this may seem like an important omission, it is worth noting that RPC calls should always receive a reply, so the number of replies can be taken as equal to the number of calls. This, of course, assumes that the server is operating correctly. A more important omission is the lack of an RPC reply counter. Such a counter could give some indication as to the similarity between the number of call and reply messages.

Nfswatch is in the public domain and is readily available as source code.

2.2.4 Tcpdump

Tcpdump can be used to output the headers of packets captured from the network interface (Jacobson et al., 1992). *Tcpdump* listens to all traffic and simply copies the headers to the display.

The output of *tcpdump* appears in protocol specific format. This makes it difficult to use as:

- Those who are not familiar with the protocol they are watching will have little or no chance of deciphering the output.

- The output is not consistent across protocols, so the knowledge of one may not help in the understanding of another.

Figure 2 shows an example of a tcpdump execution:

```
merckx.1810dbf > holly.nfs: 116 lookup fh 369557504.16777216.2560.1918173184 "ro37.rin"
holly.nfs > merckx.1810dbf: reply ok 128
merckx.1810dc0 > holly.nfs: 104 getattr fh 369557504.16777216.2560.659947520
holly.nfs > merckx.1810dc0: reply ok 96
decimus.af89e02c > lillee.nfs: 120 read fh 0,0/2560.-1711144960 4096 (4096) bytes @ 221184
```

Figure 2: Sample *tcpdump* output.

The above output is that produced for five NFS packets, captured with the command `tcpdump -c 10 -N -s 2000 -t | grep nfs`. Knowledge of the NFS protocol makes this output quite easy to read:

- `merckx` asked `holly` to *lookup* the file `"ro37.rin"` in the directory whose file handle is `3695`...`3184`. This message was `116` bytes long with a message id of `1810dbf`.

- `holly` replied with `128` bytes of data, presumably the file handle of `"ro37.rin"`.

- `merckx` asked `holly` for the *attributes* of the file with file handle `3695`...`7520` (`"ro37.rin"`), with a `104` byte message.

- `holly` returned (again presumably) the attributes of this file.

- `decimus` sent a `120` byte message to `lillee`, asking it to *read* `4096` bytes from the file whose file handle is `0,0/2`...`4960`, starting from offset `221184` in the file.

The word *presumably* appears in the reply messages above as we don't know for certain the outcome of the request. The `reply ok` in the message means the RPC call was successful — it doesn't give any indication as to the outcome of the NFS call. This is where more understanding of the protocol is needed to understand what is happening. i.e A successful *lookup* call will generally return 128 bytes of data (24 bytes of RPC header plus 104 bytes of NFS results), so it can be assumed that this call was successful. It is doubtful whether a user unfamiliar with the NFS protocol could understand what is happening.

Other points about tcpdump are:

- *Tcpdump* can read and write the packet trace to files in the Network General *Sniffer* format.

This is an excellent feature as it allows for the portability of the data. A user can capture packets with the convenience of *tcpdump*, and then analyse them with another tool. (e.g. the *Sniffer*).

• Powerful filtering capabilities are included, but are difficult to use for the beginner. As an example, it was much easier to grep through the output for "`NFS`" than use the filters.

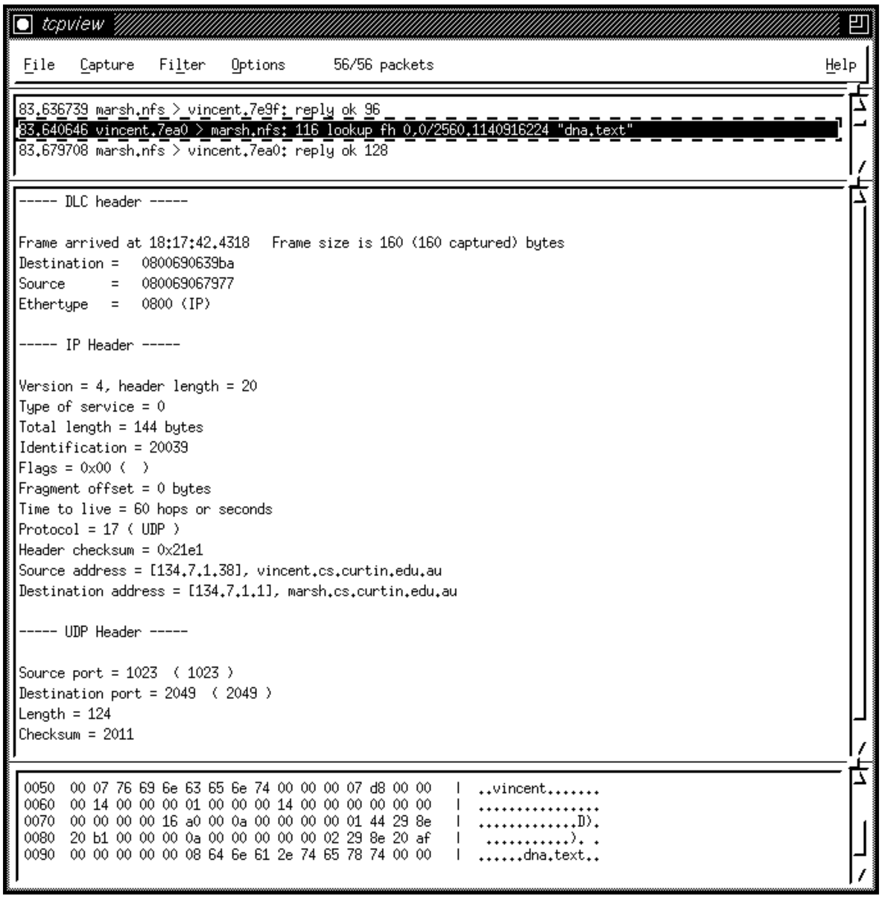
• It is in the public domain, and is available for most platforms.

Tcpdump is a classic example of a "Power User's" tool. People that know what they are doing can accomplish almost anything with it. Unfortunately, the cryptic display and difficult user interface make it almost impossible for novices.

2.2.5 Tcpview

Tcpview, which shares many features with *tcpdump* (Hunt, 1992a), is a successful attempt to improve *tcpdump's* interface.

Tcpview places an X11 "shell" around *tcpdump*, using the *Motif* look and feel. The display, as shown in figure 3 below, is segmented into three horizontal panes (as per *Sniffer* and *Analyzer*).



The description of the packets in this pane is identical to the output produced by *tcpdump*.

This pane breaks the packet down into its lower levels. In this example the DLC (ethernet), IP and UDP headers have been decoded.
Tcpview cannot decode RPC and NFS packets.

This pane gives a simple hex dump of the packet, which can be used to look at ASCII data.

Figure 3: *tcpview* decoding an NFS packet.

Tcpview adds an X interface to the filters of *tcpdump*. When the user wishes to set a filter, a simple to use dialog appears, in which complex filters can be specified.

Tcpview adds packet decoding to *tcpdump*. The data from packets is displayed in the centre pane of the display. Ethernet, IP and UDP headers are decoded as well as some protocols (e.g. ICMP, ARP). The formatting of the output still leaves much to be desired however — after the capture there is no reason why time cannot be taken to improve the output. Also, there is now time to match RPC call and reply packets. These two suggestions are obvious extensions that would make a good tool better.

Tcpview is also in the public domain, and is available from many ftp sites.

3.0 Packetman v1.0

3.1 Overview

Packetman (Benko, 1992), a retrospective X11 based protocol analyser, was developed in 1992 at Curtin University by George Benko as part of the *Netman* (Schulze et. al., 1993) project. It is available as a binary distribution for *Ultrix* and *SunOS* environments. *Packetman* promiscuously reads packets from the network interface and copies them into an internal buffer. The packets stored in this buffer can then be analysed at will.

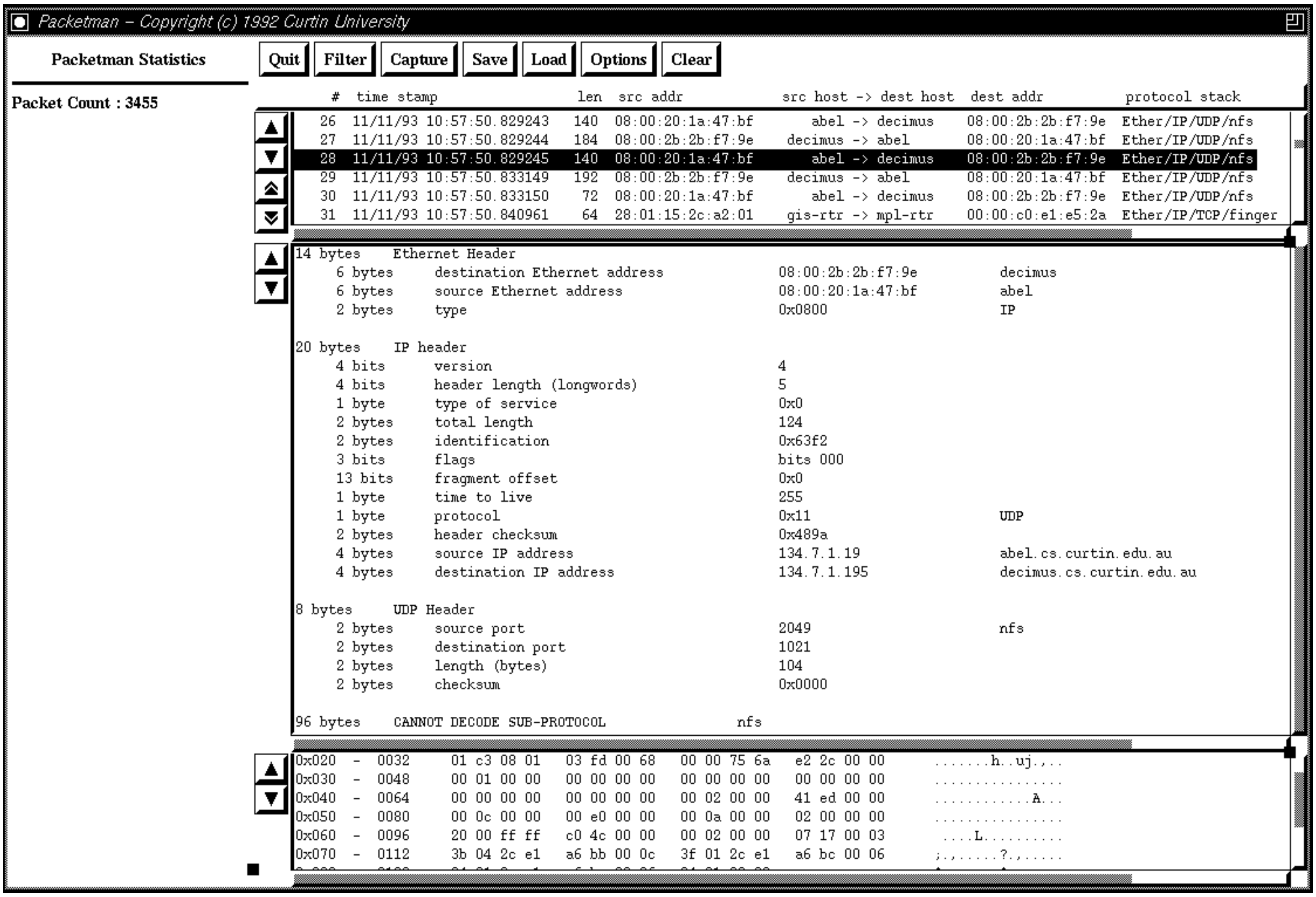


Figure 4: The *packetman* display.

As shown in figure 4, the *packetman* display is divided into four main sections. The three horizontal panes serve the same purpose as those used in the *Sniffer*, *tcpview* and *Analyzer*. The vertical pane is used to hold statistics about the packet capture. In this version of *packetman* the only statistics kept are the packet count. There are many widgets that can be used to control the actions of *packetman*, figures 5 through 9 show these controls:

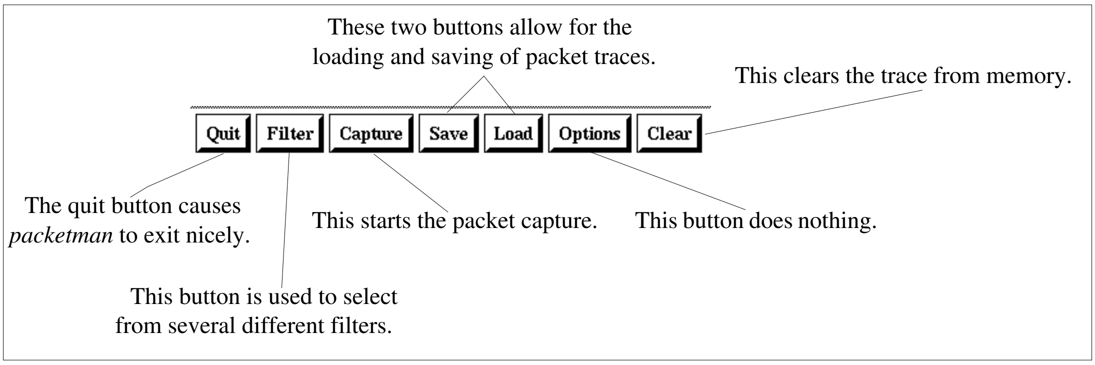


Figure 5: The main *packetman* controls.

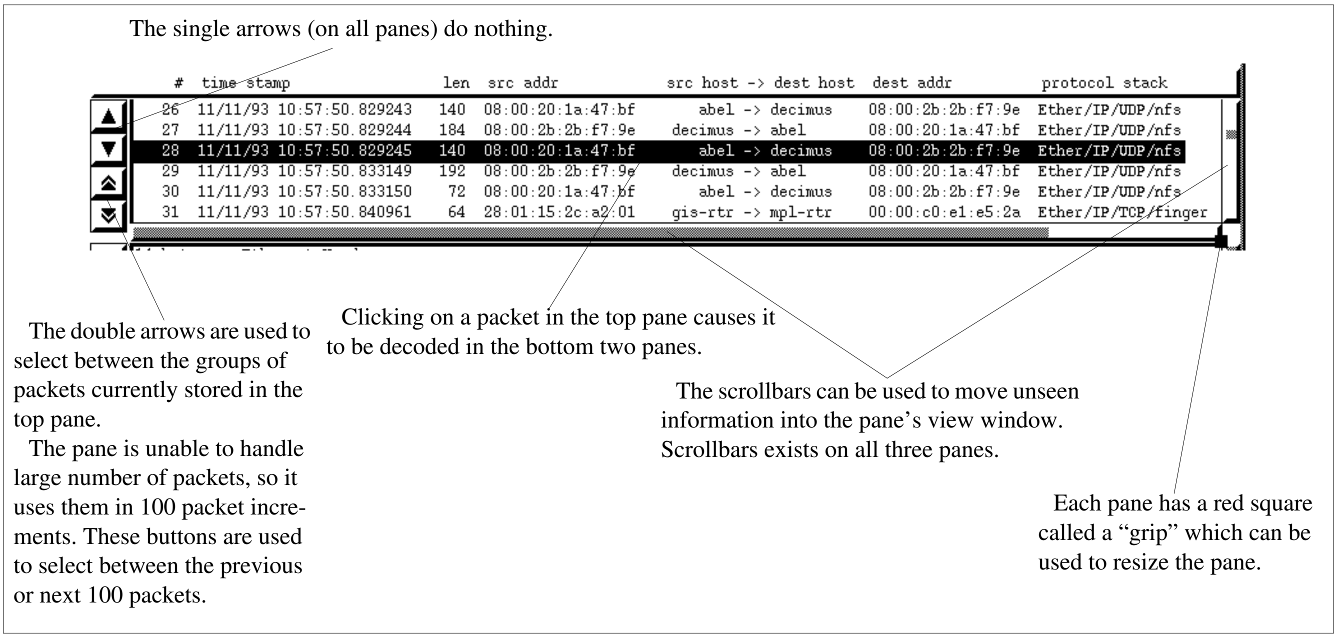


Figure 6: Packet trace pane.

The top pane displays summary information about each packet stored in the internal buffer. This information includes: The packet number, time stamp, length, source, destination and the protocol stack.

```
20 bytes    IP header
     4 bits      version                        4
     4 bits      header length (longwords)      5
     1 byte      type of service                0x0
     2 bytes     total length                   124
     2 bytes     identification                 0x63f2
     3 bits      flags                          bits 000
     13 bits     fragment offset                0x0
     1 byte      time to live                   255
     1 byte      protocol                       0x11          UDP
     2 bytes     header checksum                0x489a
     4 bytes     source IP address              134.7.1.19    abel.cs.curtin.edu.au
     4 bytes     destination IP address         134.7.1.195   decimus.cs.curtin.edu.au

8 bytes     UDP Header
     2 bytes     source port                    2049          nfs
     2 bytes     destination port               1021
```

Figure 7: Packet decode pane.

The decoding of packets takes place in this pane. Ethernet and IP headers are fully decoded, with host names and port numbers resolved where possible. The output is formatted so it can be easily read and understood.

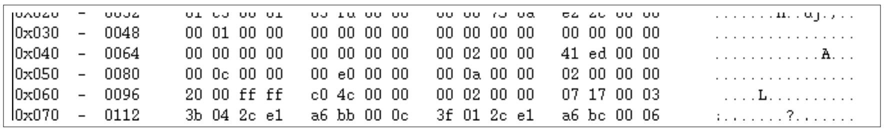


Figure 8: Packet dump pane.

The dump pane displays the packets in hexadecimal and ASCII format. The data is printed 16 bytes per line and is formatted for easy reading.

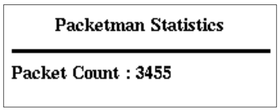


Figure 9: Statistics pane.

The statistics pane shows the current packet count, which is updated at the end of each capture or load operation.

3.2 Mechanism

3.2.1 Capturing Packets

3.2.1.1 Network Interfaces

As mentioned earlier, *packetman* places the network interface into promiscuous mode, in order to watch all packets on the network. Normally, an interface will copy a packet from the network and pass it on to the operating system only if the packet is destined for that machine. Packets can be destined for a machine in two ways:

- The packet is specifically addressed to the machine. e.g. In an ethernet LAN the destination address in the ethernet header is the same as the address of the machine.

- The packet is broadcast. i.e. The packet is addressed to all machines on the network. In ethernet this occurs when the destination address is `ff:ff:ff:ff:ff:ff`.

When in promiscuous mode, the interface will pass on all packets, regardless of their destination. It is important that interfaces perform their functions as efficiently as possible. If the system is too slow then packets will be missed. Such packets are referred to as dropped.

When capturing, packets are read from the interface in chunks. A chunk of packets refers to the number of packets returned from each interface read. Reading in chunks is more efficient that reading one packet at a time. The number of packets in each chunk will differ between reads.

Packetman uses one of two interfaces, depending on the environment:

3.2.1.1.1 SunOS: Network Interface Tap (NIT)

NIT is composed of three streams modules and drivers (Sun, 1990), which together provide the facilities of the network interface. The modules are:

- *nit_if(4M)*: This is the streams device driver which interacts with the system's ethernet drivers. An instance of this device is bound to an ethernet interface (e.g. `le0`) and can then be used to send and receive packets.

• *nit_pf(4M)*: This provides packet filtering capabilities so that unwanted packets can be discarded.

• *nit_buf(4M)*: This allows for the batch read of packets instead of reading them one at a time. It is more efficient to batch read when large numbers of packets are being taken from the interface.

NIT supports promiscuous mode monitoring (for the super-user), but this is limited by the fact that it cannot monitor its own outbound traffic.

Unfortunately, NIT is inefficient. McCanne and Jacobson (1993) explain that NIT's inefficiency comes mainly from its filtering module, *nit_pf(4M)*. NIT copies packets from the interface before filtering them, which leads to inefficiencies. In contrast, the BSD Packet Filter (BPF) (McCanne, 1993), filters packets in place; i.e. Before copying them from the interface. BPF is up to 100 times faster than NIT.

BPF will be examined later in this report.

3.2.1.1.2 Ultrix: Packetfiter

The Ultrix *packetfilter(4)* (DEC) follows a different approach to NIT in that it is not composed of separate modules. The packet filter consists of kernel resident code which appears to the user as a set of character device drivers (`/dev/pf/pfilt`*`nnn`*). To use *packetfilter*, the user opens one of the drivers using the library routine *pfopen(3)*.

Packetfilter provides for batched reads, filtering and also promiscuous mode operation. An additional feature over NIT is that *packetfilter* allows any user (not just the super-user) to use the interface in promiscuous mode. The super-user can configure this option by using *pfconfig(8c)*.

3.2.1.2 Storing Packets

Once a chunk has been read from the interface, it's processed and packets are copied from it into *packetman's* internal buffer. Every packet in the chunk is subjected to the following two processes:

3.2.1.2.1 Filtering

Most network interfaces include filtering mechanisms, which filter packets quite efficiently. In the *Netman* project it was decided that they would not be used. It was felt that the portability of code gained by implementing filters was more important than the efficiency gained by using the inbuilt filters (Schulze et. al., 1993).

The packets are checked against *packetman's* filters, to determine whether they're to be kept or discarded. The filters are set by the user at run-time, by selecting the protocols wanted from the *filter* menu.

The filters are implemented as a linked list, with each element containing the filter definition plus the pointer to the next element in the list. Every packet is checked against every filter until either:

- The packet is matched. The routine will exit at the first match found, as no other checks will be necessary. The lowest level details are checked first. i.e. The ethernet type, followed by any sub-protocols etc.

- All filters have been used.

3.2.1.2.2 Copying

A matched packet will be copied into *packetman's* three megabyte buffer, which will store a maximum of 10, 000 packets. These packets are access by an array of type `Packet`, which holds the address of each packet in the buffer. Figure 10 shows the declarations of this storage.

```
#define KILOBYTE 1024
#define MEMORY_SIZE (3*KILOBYTE*KILOBYTE)
static char memory_buffer[MEMORY_SIZE];

typedef struct {
     int len;
     struct timeval time;
     char *ptr;
} Packet;
Packet packet_index[MAX_PACKETS]; /* pointer to all the packets */
```

Figure 10: Internal buffer definitions.

The `Packet` type holds the length and timestamp of the packet as well as the pointer to its offset in the memory buffer.

3.2.2 Displaying Packets

3.2.2.1 List Widgets

As stated previously, the packets are displayed in their various forms inside the three horizontal panes. To be more precise, however, the packet information is displayed using *viewlist* widgets. A viewlist widget is used to display many lines of textual data. The widget contain scrollbars which can be used to view data not currently in the widget's view window.

After the capture or load is completed, the packet trace viewlist is filled with the first group of packets. This grouping is necessary as the list widget can't store all 10, 000 packets — it will become unresponsive and may cause *packetman* to fail. This limitation is overcome by the and buttons. These allow the user to select between the next or previous group of packets.

Displaying the packet summaries in the viewlist is quite simple. A string for each packet is created by the protocol decoding routines, and is placed into a structure (`StringList`) which holds all such strings. These strings are then "assigned" to the viewlist and it is redisplayed for the new information to take effect.

3.2.2.2 Packet Decode List

This operates similarly to the packet trace list.

The strings are generated by the various decode routines and are added to the `StringList` structure. Several support routines are used to accomplish this. When the decoding of the packet is completed the viewlist is redisplayed.

3.2.2.3 Packet Dump List

The output for the dump list is created by the one function, which fills the `StringList` structure with the output.

3.2.3 Decoding Packets

3.2.3.1 Overview

In *packetman*, there are two levels to which packet decomposition takes place. These are referred to as:

- Half decoding: This refers to the identification of the protocol stack of each packet as displayed in the packet trace viewlist.
- Full decoding: This refers to the full packet decomposition which takes place in the packet decode viewlist.

To decode a packet, it is necessary to step through its data and identify the various protocols encountered until a terminating protocol is found[1]. The group of protocols present in a packet is referred to as the protocol stack. For example, the protocol stack of a typical NFS packet is: Ethernet, IP, UDP, NFS.

Fortunately most protocols have, if not standard, well known numerical identifiers which can be used to distinguish them from other protocols. A table of these values can be used against the packet data to identify the various protocol types. Decomposition is then a process of decoding one level of a packet, identifying the next level and then decoding that, if possible. This is the approach used in *packetman*.

To fully explain the decoding process, an example is required. The following section shows how *packetman* decodes a telnet packet.

3.2.3.2 Decoding a Telnet Packet

This description will not go into specific detail as to the format of a telnet packet, or its ethernet/IP/TCP headers. For a detailed description see Comer (1991).

1. A terminating protocol is one which has no sub-protocols. For example, TCP is not a terminating protocol but telnet is.

3.2.3.2.1 Ethernet Header

The ethernet header is first decoded. This header consists of three fields: The destination address, source address and the frame (protocol) type. The frame type is then used to look up the protocol/type table to determine:

- If the protocol can be identified.
- If so, the routine to be used to decode it, if one exists.

```
14 bytes    Ethernet Header
     6 bytes     destination Ethernet address          00:00:b5:04:04:2d        VIMC1
     6 bytes     source Ethernet address               08:00:20:1a:47:bf        abel
     2 bytes     type                                  0x0800                   IP
```

Figure 11: Telnet ethernet frame.

For a telnet packet, the frame type will be `0x800`, identifying it as an IP packet.

3.2.3.2.2 IP Header

Similarly to the ethernet header, the IP header is decoded. It consists of many more fields than the ethernet header, including: Length, identification, flags, time to live, protocol type and the source and destination IP address — which may or may not be the same as the ethernet addresses[1]. The protocol type is then used for the table look up, and the next routine is called.

```
20 bytes    IP header
     4 bits      version                          4
     4 bits      header length (longwords)        5
     1 byte      type of service                  0x0
     2 bytes     total length                     498
     2 bytes     identification                   0x1e2d
     3 bits      flags                            bits 000
     13 bits     fragment offset                  0x0
     1 byte      time to live                     60
     1 byte      protocol                         0x6                  TCP
     2 bytes     header checksum                  0x4ff1
     4 bytes     source IP address                134.7.1.19           abel.cs.curtin.edu.au
     4 bytes     destination IP address           134.7.1.199          VIMC1.cs.curtin.edu.au
```

Figure 12: Telnet IP header.

For telnet, the protocol type will be either `6` (TCP) or `17` (UDP). Telnet usually uses TCP.

3.2.3.2.3 TCP Header

The TCP header includes the source and destination ports, sequence number as well as several other fields.

TCP is connection orientated — each packet is part of a conversation between two machines. The points of communication on each machine are the two ports included in the header, which each represent some application program. As there are two protocol identifiers this introduces a problem of precedence — which port number should be used for the table look up? A similar problem exists for UDP

1. This is because of the presence of routers and bridge on the network. If a packet from the other side of a router is received by a machine, then the source ethernet address will be that of the router. The source IP address, however, will be that of the originator of the packet.

In practice, fortunately, only one of the port numbers is generally recognised, so no conflict comes about.

```
20 bytes    TCP Header
     2 bytes     source port                     23                     telnet
     2 bytes     destination port                2611
     4 bytes     sequence number                 392264901
     4 bytes     acknowledgement number          4111465074
     4 bits      header length (longwords)       5
     6 bits      reserved                        0
     6 bits      flags                           bits 010000
     2 bytes     window                          4096
     2 bytes     checksum                        0x3070
     2 bytes     urgent pointer                  0x0
```

Figure 13: Telnet TCP header.

For telnet, the port number is `23`, which is used for both TCP and UDP.

3.2.3.2.4 Telnet Data

The final stage of the decoding operation is that of the telnet packet itself.

```
458 bytes   telnet packet
     172 bytes   data                      "     .[4mFirst.[m .[4mletter.["  . . .
     2 bytes     command sequence          255 17                   Command 17
     131 bytes   data                      "h....*.........p.....0........"  . . .
     2 bytes     command sequence          255 8                    Command 8
     105 bytes   data                      "...?...i......................"  . . .
     2 bytes     command sequence          255 17                   Command 17
     44 bytes    data                      ".........*.....h.....0........"  . . .
```

Figure 14: Telnet data.

Protocols have set syntax, So if there rules are known they can be decoded.

3.3 Evaluation

The original release of *packetman* provided the basis for an excellent protocol analyser. It needs much work to make it as useful as some other tools available. Some points about *packetman* worth mentioning are presented below:

3.3.1 Strengths

- *Packetman* presents the user with a simple, easy to use interface that enables him/her to analyse the traffic on an ethernet LAN. The output of decoded packets is well formatted and is pleasing to the eye.

- IP, UDP, TCP headers and several protocols are decoded, with host names resolved from IP and ethernet addresses where possible.

- *Packetman* is available free as a binary distribution for DEC/Ultrix and Sun/SunOS environments.

3.3.2 Deficiencies

While *packetman* is a good tool, it is not perfect. Some criticisms are:

- Executing *packetman* with garbage arguments will result in a core dump.

- X events are not processed during the load, save and capture operations. This causes several unwanted conditions:
    - The counter is not updated whilst these processes are taking place. This gives the user no feedback as to what is happening.
    - It is not possible to cancel the operations in any way, as no controls can be used.
- The X interface is poorly laid out. There are two main problems with this:
    - The buttons used to select between different groups of packets in the top pane are on the other side of the window to the scroll bars. This results in much mouse movement which becomes, over a period of time, inconvenient and annoying.
    - The vertical statistics pane must be resized to zero width in order to see the full amount of information in the top pane. This is not a problem until a new capture or load takes place. Because the resize obscures the counter, the pane becomes its default size once more, and must be resized again. This can become annoying.
- The files produced by *packetman* are not in the Network General *Sniffer* format. This means that the traces produced by *packetman* cannot be used elsewhere, and *packetman* cannot load traces captured elsewhere.
- The filter set still leaves much to be desired. Though they are certainly easy to use, they lack the power of other filter sets, such as those included in *tcpdump/tcpview*. Most important of these deficiencies is the lack of host filtering. i.e. Isolating communication between particular hosts.
- Currently, the set of recognisable protocols is too small to make *packetman* of much use. IP, UDP and TCP headers are fully decoded, but few protocols are taken to their maximum level. *Packetman* recognises most protocols from the type fields in the headers, but cannot proceed with any decoding.

4.0 Facilities Used

4.1 Workstations

The development of this project originally commenced on DECstation 5000 series architectures, as the Ultrix *packetfilter(4)* allowed promiscuous mode operation for non super-users.

When porting to Sun SPARCstation architectures commenced, super-user (*using sudo(8)*) access was allowed on a non-secure machine, to enable testing to take place. Henceforth, it was decided to switch all development to Sun, as the compilation times were significantly faster than the DEC machines. Table 1 shows the compilation times on the various architectures available for use.

Machine	Architecture	Time for `xmkmf -a` and `make`
decimus	Decstation 5000/240, 32 Mb	0:41.15 + 4:54.38 = 5:35.53
lucius	Decstation 5000/25, 16 Mb, isolated by router.	1:45.97 + 10:49.59 = 12:35.56
cain	Sun SPARCstation IPX, 64 Mb	0:20.96 + 3:43.69 = 4:04.65
fog	Sun SPARCstation IPX, 32 Mb, isolated by router.	0:35.76 + 4:04.88 = 4:40.64

Table 1: Compilation time for various architectures.

As can be seen, the compilation times for the Suns were significantly quicker than those for the DECs. It should be noted that:

- All compilations involved reading from and writing to a NFS partition mounted from *lillee*, an SGI Indigo. *Decimus* and *cain* were on the same network segment as *lillee*, whereas *lucius* and *fog* were separated by PC-based routers.

- The Suns performed two links to the DEC's one. It was necessary to link the executable with its supporting libraries both dynamically and statically

- While *decimus* was only 1:30 slower than *cain*, it was often is use for purposes unrelated to this project, and so was unavailable. This left *lucius* (and many identical machines) as the only DEC alternative.

4.2 Revision Control System (RCS)

To keep track of major changes in the source files, it was decided that RCS (Tichy, 1991) would be used. RCS is used for version control. When major changes have been incorporated into a file it can be "checked in". This stores the file as a revision of the previous version. To save disk space, revisions are stored as differences from previous versions. RCS offers many other features which help to control the development of software.

5.0 Aims

The major aims of this project are directly related to the problems discussed in section 3.3.2. These are:

- To improve the X interface.
    - Reposition the widgets which were identified as problems.
    - Process events during capture, in order to give feedback to the user.
- Decode more protocols.
    - The basic skeleton for ICMP is provided and needs to be completed so that this pro-

tocol can be added.

• NFS, and therefore RPC, is the protocol of most importance in the School of Computing — it represents over 50% of bandwidth usage.

• If time allows, other protocols will be decoded. It will be decided which other protocols to decode when the time comes.

• Improve the filtering mechanism:

• A better selection mechanism is needed for the filters. The current menu system is easy to use but lacks any feedback to the user as to the protocols selected. It also only allows the selection of the specified protocols. The user should be able to be add other protocols to the filter set.

• Host filters are required to isolate traffic. This is needed as the sheer volume of network traffic makes the process of looking for specific hosts manually almost impossible.

6.0 Changes

6.1 X11 Interface

The changes made to the X11 interface were in line with the problems outlined in section 3.3.2. While most of these changes may appear trivial, they have resulted in an interface which is easier and more pleasant to use.

6.1.1 Processing Events

The lack of event processing prevented any feedback as to the status of the capture from reaching the user, and also prevented any possibility of cancelling the operation.

The code to implement event processing is trivial:

```
XtInputMask Xmask;

/* process events */
while(Xmask = XtAppPending(appContext)) {
     XtAppProcessEvent(appContext, Xmask);
}
```

The positioning of this code in the capture sequence was important, in order to give the best event processing. There were several positions where it could be placed:

• Immediately before or after the read from the network interface.

• Within `UpdateStatisticsWidgets`, the function which updates the counter value.

• Within the function `CopyPacket`, which copies wanted packets from the read buffer into the internal buffer. i.e Performs the operations described in section 3.2.1.2.

The position before or after the read was immediately rejected, as events would not be processed while load operations are taking place. The update frequency would also be unsatisfactory, especially when a large number of small packets are read from the interface.

The logical decision, then, seemed to be to place the code in `UpdateStatisticsWidgets`, which would ensure the counter was updated each time its value was changed. This placement was implemented, but brought another problem to the foreground — expose events were being processed too infrequently, especially when many packets were being filtered out.

This necessitated the move of the code into `CopyPacket`. With events processed for every packet read, all types of events were processed to an acceptable degree.

6.1.2 Repositioning Widgets

6.1.2.1 Arrows

The arrow widgets assigned to each pane were moved to the right hand side of display. This involved altering the definitions of the widgets in the applications resource file from

```
Packetman.widgets:\
…     …     …
…     …     …
*xpViews                   Paned      packetView \
*packetView                Box        packetButtons \
*packetButtons             Command    upButton \
*packetButtons             Command    downButton \
*packetButtons             Command    prevButton \
*packetButtons             Command    nextButton \
*packetView                Viewport   packetPort \
*packetView.packetPort     List       packetList \
*xpViews                   Paned      decodeView \
*decodeView                Box        decodeButtons \
*decodeButtons             Command    upButton \
*decodeButtons             Command    downButton \
*decodeView                Viewport   decodePort \
*decodeView.decodePort     List       decodeList \
*xpViews                   Paned      dumpView \
*dumpView                  Box        dumpButtons \
*dumpButtons               Command    upButton \
*dumpButtons               Command    downButton \
*dumpView                  Viewport   dumpPort \
*dumpView.dumpPort         List       dumpList
```

to

```
Packetman.widgets:\
…     …     …
…     …     …
*xpViews                   Paned      packetView \
*packetView                Viewport   packetPort \
*packetView.packetPort     List       packetList \
*packetView                Box        packetButtons \
```

```
*packetButtons            Command    upButton \
*packetButtons            Command    downButton \
*packetButtons            Command    prevButton \
*packetButtons            Command    nextButton \
*xpViews                  Paned      decodeView \
*decodeView               Viewport   decodePort \
*decodeView.decodePort    List       decodeList \
*decodeView               Box        decodeButtons \
*decodeButtons            Command    upButton \
*decodeButtons            Command    downButton \
*xpViews                  Paned      dumpView \
*dumpView                 Viewport   dumpPort \
*dumpView.dumpPort        List       dumpList \
*dumpView                 Box        dumpButtons \
*dumpButtons              Command    upButton \
*dumpButtons              Command    downButton
```

The above entries from the resource file specify the widget hierarchy for the application. The bold entries indicate the before and after states of the button definitions. The first bold area in the "from" entries says that:

> upButton, downButton, prevButton, nextButton are Command widgets whose parent is packetButtons. PacketButtons is a Box widget whose parent is packetView, which is a Paned widget whose parent is xpViews.

The widgets are created in the order that they appear in definitions. This causes the packetButtons to be created before the packetPort and packetList. It is for this reason that the buttons appear on the left hand side of the viewlist. By moving the definitions for the buttons after the viewlist, they are created last and therefore appear on the right hand side of the display.

6.1.2.2 Counter

To remove the statistics pane, it was necessary to relocate the counter. The obvious position for this was the top of the display, in-line with the various buttons. At the same time, the opportunity was taken to add the current/total capture statistics (the totalCount widget) along side the packetCount widget.

To change the position of the widget, its path in the applications resource file was altered, for similar reasons to the alterations made in section 6.1.2.1. The path was changed to reflect the new position of the widget at the top of the display. The implementation of the totalCount widget was a case of duplicating the packetCount widget code and definitions and modifying them were necessary:

```
root                      Paned      xpViews \
*xpViews                  Form       xpButtons \
*xpButtons                Command    quitButton \
…     …     …
…     …     …
*xpButtons                Command    clearButton \
*xpButtons                Form       xpStats \
*xpStats                  Label      packetCount \
*xpStats                  Label      totalCount \
…     …     …
```

...   .

In addition to changing its position, the counter's frequency of update was changed to give more useful feedback. Instead of increments at intervals of 50, it was decided to have a floating increment as shown in table 2:

Counter Value (x)	Increment Value
x 50	1
50 < x 100	10
x > 100	50

Table 2: Counter increment values.

6.1.2.3 Statistics Pane

The removal of the statistics involved removing all references to it in the applications resource file, as well as its creation in the source code. After its removal, the width of the three horizontal panes was increased to take into account the space made available. The width the panes was increased by altering their values in the resources file:

```
Packetman*xpViews.width: 1110
```

This results in a display which, while still the same size as the original, shows more information.

6.1.3 Dialogs

6.1.3.1 Capture Dialog

The capture dialog accepts an integer from the user which is the number of packets to be read from the network interface. There are two main deficiencies with this dialog:

- There is no default value. The user must always enter a value, which can become frustrating.

- There is no provision for the cancellation of a running capture.

The first problem was solved relatively easily. If an invalid entry is received, be it characters or a number too large, the capture will default to 10, 000 packets or the number of packet entries available in the buffer. This makes the process less "fiddly".

The most important problem was the lack of a cancellation facility. This was overcome by the addition of a capture flag — while it is true the capture will proceed, but will stop as soon as it is set to false. The flag is set true immediately before the capture commences (i.e. it is set in the callback for the start button) and can be set to false in one of two ways:

- The capture ends normally. The flag is only true while a capture is taking place.

• The user clicks the stop button. This button replaces the start button in the dialog once the capture commences. The callback for this button unsets the capture flag.

The return key is bound to both the start and stop button, so it can be used instead of the mouse. This makes the use of the dialog more natural.

6.1.3.2 File Dialogs

The load and save dialogs were modified to allow the operation to be cancelled any time after it commences. This was accomplished by checking the capture flag during the operations. Similarly to the capture dialog, the return key was mapped to the start/stop buttons.

6.1.3.3 Host Dialog

The host dialog is the interface to the host filters, which were implemented to isolate hosts:

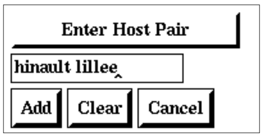


Figure 15: Host filtering dialog.

To use the host filtering, two host names are entered in the dialog, separated by white space. The *Add* button is used to add the tuple to the host filters. The *Clear* button clears all the filters and the *Cancel* button closes the dialog box. The host filters are implemented in a way similar to that of the protocol filters described in section 3.2.1.2.1.

Figure 16 shows the final *packetman* display.

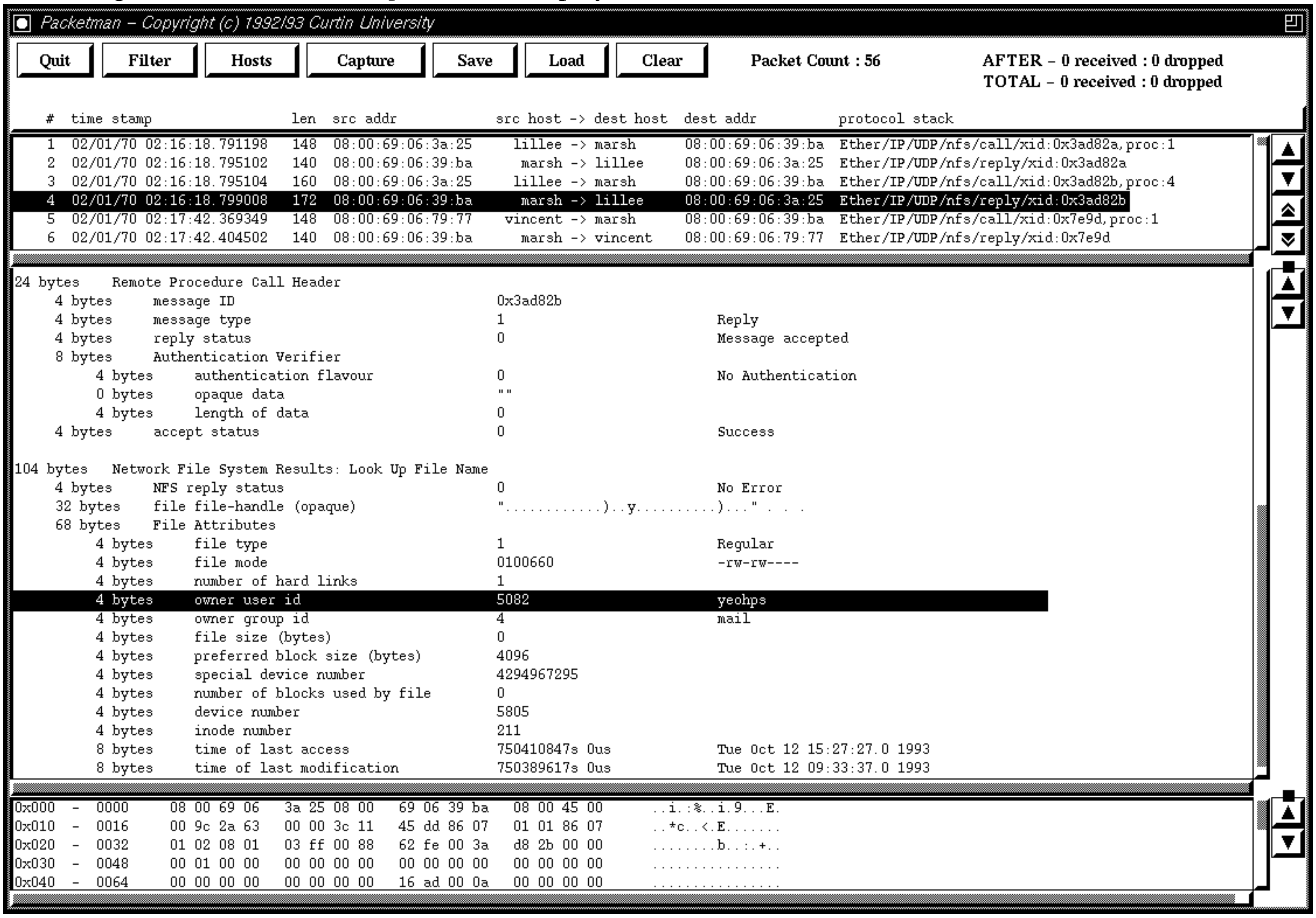


Figure 16: *Packetman* v1.1 display.

6.2 Arguments

Packetman died when it was executed with garbage arguments. This problem occurred because of the use of the *XtDestroyApplicationContext(3Xt)* routine, which should only be used to destroy X11 applications.

When garbage arguments were entered, *XtDestroyApplicationContext* was used to destroy *packetman*. Unfortunately, *packetman* had not yet been initialised as an X11 application using the routine *XtVaAppInitialize*, and so a crash occurred. This problem was fixed by replacing the call to *XtDestroyApplicationContext* with a call to *exit(3)*.

6.3 Sniffer Files

The ability to read and write *Sniffer* files is essential for a protocol analyser to be successful. This allows the portability of files between applications, so that the strengths of some can be used to overcome the weaknesses of others.

Sniffer file compatibility could of been provided in one of two ways: by writing the code to perform the required operations, or by using existing code. The second option was obviously preferable.

Tcpview, described in section 2.2.5, contains public domain code which performs these operations. This code (Hunt, 1992b) was modified slightly and linked in with the compilation. The load and save functions were modified to use the functions provided by this source code.

6.4 Protocols

6.4.1 Internet Control Message Protocol (ICMP)

6.4.1.1 Overview

IP is a connectionless-mode protocol, and therefore does not contain any error reporting or correcting mechanisms (Black, 1992). To provide these services, ICMP (Postel 1981, Mogul and Postel 1985) which provides for these basic error correcting and administrative messages is used. Black (1992) identifies several key aspects of ICMP:

- ICMP uses IP to deliver its messages and IP must use ICMP i.e ICMP is a necessary requirement for the successful implementation of IP.
- ICMP doesn't make IP reliable — it simply reports errors, and leaves it to higher level protocols to correct them. The messages returned can be acted on or ignored depending upon the host's choice.
- ICMP will not send messages regarding ICMP packets. Such messages are not sent to prevent infinite loops from occurring.

6.4.1.2 Message Types

There are several message types for ICMP, all of which have a similar structure.

6.4.1.2.1 Destination Unreachable Message

Type (3)	Code (0-5)	Checksum
Unused		
Internet Header + 64 bits of Data Datagram		

Figure 17: Destination Unreachable message format.

The `type` field indicates which type of ICMP message this is, in this case destination unreachable. These messages are sent when a packet cannot be delivered to its destination. The reason for this failed delivery corresponds to the `code` field, the values of which are defined below:

Code	Name	Description
0	Network unreachable	If the destination network cannot be reached from a gateway, the gateway can send a destination unreachable message.
1	Host unreachable	A gateway may be able to determine that the host is unreachable, in which case a message can be sent.
2	Protocol unreachable	If IP cannot deliver the message because the destination protocol is not active a message may be sent.
3	Port unreachable	If the indicated port is not active, then a destination unreachable message can be sent.
4	Fragmentation needed and DF set	If the don't fragment bit of message is set, and the packet cannot be delivered without being fragmented, it will be discarded and a destination unreachable sent to the source host.

The `unused` field is unused and should be zero. The `checksum` is used to ensure the packet was delivered correctly. The message also includes the `internet header and first 64 bits` of the packet that caused the ICMP message to be sent. This is so receiver of the ICMP message can determine which process needs to be notified of the problem, so that the correct action can be taken.

```
36 bytes    ICMP Packet - Destination Unreachable Message
     1 byte      type of message                          3                          Destination Unreachable
     1 byte      type sub code                            3                          Port Unreachable
     2 bytes     checksum                                 0x4919
     4 bytes     unused (must be zero)                    0
     28 bytes    IP header + 64 bits of Original Data Datagram

20 bytes    IP header
     4 bits      version                                  4
     4 bits      header length (longwords)                5
     1 byte      type of service                          0x0
     2 bytes     total length                             76
     2 bytes     identification                           0x396c
     3 bits      flags                                    bits 000
     13 bits     fragment offset                          0x0
     1 byte      time to live                             29
     1 byte      protocol                                 0x11                       UDP
     2 bytes     header checksum                          0x101e
     4 bytes     source IP address                        134.7.1.195                decimus.cs.curtin.edu.au
     4 bytes     destination IP address                   134.7.70.70                cujo.curtin.edu.au

8 bytes     UDP Header
     2 bytes     source port                              108                        snagas(SNA Gateway Access Server)
     2 bytes     destination port                         22877
     2 bytes     length (bytes)                           0
     2 bytes     checksum                                 0x0020
```

Figure 18: Destination Unreachable/Port Unreachable message.

6.4.1.2.2 Time Exceeded Message

Type (11)	Code (0-1)	Checksum
Unused		
Internet Header + 64 bits of Data Datagram		

Figure 19: Time Exceeded message format.

A time exceeded message may be sent in one of two cases:

Code	Name	Description
0	Time to live exceeded in transit	When a packet is sent from the source host, the TTL field is set to some value. At each stage of the packet's journey (i.e. each gateway or host encountered) the TTL field is decremented by one. If this field is ever zero, the packet is discarded and a time exceeded message is sent.
1	Fragment reassembly time exceeded	If an IP packet is large, it may need to be fragmented into several pieces, each of which is forwarded to the destination. The destination starts a timer when it receives the first fragment of the packet, and will assemble all the fragments into the original packet. If the timer times out before the packet is reassembled, a time exceeded message is sent

Traceroute(8) (Jacobson, 1989), uses the TTL field to record the route of an IP packet. It sends meaningless UDP packets to the destination with increasing from zero TTL values. This causes timeouts along the packet's route, of which the source is notified. From these timeouts the route of the packet can be determined.

```
     4 bytes     source IP address               139.130.4.5          usa.gw.au
     4 bytes     destination IP address          134.7.1.23           hinault.cs.curtin.edu.au

36 bytes    ICMP Packet - Time Exceeded Message
     1 byte      type of message                 11                   Time Exceeded Message
     1 byte      type sub code                   0                    Time to Live Exceeded in Transit
     2 bytes     checksum                        0xe0ae
     4 bytes     unused (must be zero)           0
     28 bytes    IP header + 64 bits of Original Data Datagram
```

Figure 20: Time Exceeded/TTL Exceeded Message.

6.4.1.2.3 Parameter Problem Message

Type (12)	Code (0)	Checksum
Pointer	Unused	
Internet Header + 64 bits of Data Datagram		

Figure 21: Parameter Problem message format.

This message is sent when a packet is discarded due to errors in its header parameters. The `pointer` field in the message identifies the octet where the error occurred.

6.4.1.2.4 Source Quench Message

Type (4)	Code (0)	Checksum
Unused		
Internet Header + 64 bits of Data Datagram		

Figure 22: Source Quench message format.

A source quench messages will be sent by a gateway when it has run out of buffer space (and therefore cannot process the packet) or by a host when packets are arriving too fast to be processed. On receipt a source quench message, the source host should slow down the rate at which it is sending packets, and then slowly increase this rate until normal speed is reached or another source quench is received. Most hosts and gateways will send source quenches shortly before their facilities are overloaded.

```
     4 bytes     source IP address                 134.178.3.251         HEAD_OFFICE.BoM.GOV.AU
     4 bytes     destination IP address            134.7.70.149          cupid.curtin.edu.au

36 bytes    ICMP Packet - Source Quench Message
     1 byte      type of message                   4                     Source Quench Message
     1 byte      type sub code                     0
     2 bytes     checksum                          0x7fdd
     4 bytes     unused (must be zero)             0
     28 bytes    IP header + 64 bits of Original Data Datagram
```

Figure 23: Source Quench message.

6.4.1.2.5 Redirect Message

Type (5)	Code (0-3)	Checksum
Gateway Internet Address		
Internet Header + 64 bits of Data Datagram		

Figure 24: Redirect message format.

A redirect message is sent to the source if a gateway on the packet's route discovers a shorter path to the destination. The `gateway internet address` identifies to the source host the internet address of the gateway to which future packets should be sent.

```
     4 bytes     source IP address                 134.7.70.100          aarnet-gw.curtin.edu.au
     4 bytes     destination IP address            134.7.70.1            cc.curtin.edu.au

36 bytes    ICMP Packet - Redirect Message
     1 byte      type of message                   5                     Redirect Message
     1 byte      type sub code                     1                     Redirect Datagrams for the Host
     2 bytes     checksum                          0x7192
     4 bytes     gateway internet address          134.7.70.251          cc-gw-e0.curtin.edu.au
     28 bytes    IP header + 64 bits of Original Data Datagram
```

Figure 25: Redirect Message/Datagram For Host message.

6.4.1.2.6 Echo or Echo Reply Message

Type (8, 0)	Code (0)	Checksum
Identifier		Sequence Number
Data...		

Figure 26: Echo/Echo Reply message format.

These messages are used to check if a particular host or gateway is operational and reachable. The `type` field indicates the type of message. Type `8` is an echo request while type `0` is an echo reply. The `identifier` is used to indicate the process which is sending the messages, while the `sequence number` is used to distinguish between the message that process has sent. Echo messages can also include optional data, which must be returned in the echo reply message. *Ping(8C)* sends echo request messages to a specified host and listens for any response.

```
     4 bytes     source IP address                 134.7.1.2            lillee.cs.curtin.edu.au
     4 bytes     destination IP address            130.43.2.2           apple.com

64 bytes    ICMP Packet - Echo or Echo Reply Message
     1 byte      type of message                   8                    Echo Message
     1 byte      type sub code                     0
     2 bytes     checksum                          0xc1e5
     2 bytes     identifier                        0x6184
     2 bytes     sequence Number                   0x1
     56 bytes    optional Data                     ",.VV..f@.............J........" . . .


     4 bytes     source IP address                 130.43.2.2           apple.com
     4 bytes     destination IP address            134.7.1.2            lillee.cs.curtin.edu.au

64 bytes    ICMP Packet - Echo or Echo Reply Message
     1 byte      type of message                   0                    Echo Reply Message
     1 byte      type sub code                     0
     2 bytes     checksum                          0xc9e5
     2 bytes     identifier                        0x6184
     2 bytes     sequence Number                   0x1
     56 bytes    optional Data                     ",.VV..f@............ .1.....8" . . .
```

Figure 27: Echo/Echo reply messages.

6.4.1.2.7 Timestamp and Timestamp Reply Message

Type (13, 14)	Code (0)	Checksum
Identifier		Sequence Number
Originate Timestamp		
Receive Timestamp		
Transmit Timestamp		

Figure 28: Timestamp/Timestamp Reply message format.

These messages are used to determine the network latency between the source and destination host. The `originate timestamp` is set immediately before the timestamp message is sent. The `receive timestamp` is set immediately upon the arrival of the message at the destination. The `transmit timestamp` is set immediately before the destination sends the timestamp reply to the originating host.

```
     4 bytes     source IP address                       134.7.1.6                pride.cipal.cs.curtin.edu.au
     4 bytes     destination IP address                  134.7.1.195              decimus.cs.curtin.edu.au

28 bytes    ICMP Packet - Timestamp or Timestamp Reply Message
     1 byte      type of message                         13                       Timestamp Message
     1 byte      type sub code                           0
     2 bytes     checksum                                0xa07e
     2 bytes     identifier                              0xad
     2 bytes     sequence Number                         0x2a13
     4 bytes     originate Timestamp (ms since 0:00 GMT) 8523295
     4 bytes     receive Timestamp (ms since 0:00 GMT)   0
     4 bytes     transmit Timestamp (ms since 0:00 GMT)  0

     4 bytes     source IP address                       134.7.1.195              decimus.cs.curtin.edu.au
     4 bytes     destination IP address                  134.7.1.6                pride.cipal.cs.curtin.edu.au

28 bytes    ICMP Packet - Timestamp or Timestamp Reply Message
     1 byte      type of message                         14                       Timestamp Reply Message
     1 byte      type sub code                           0
     2 bytes     checksum                                0x8254
     2 bytes     identifier                              0xad
     2 bytes     sequence Number                         0x2a13
     4 bytes     originate Timestamp (ms since 0:00 GMT) 8523295
     4 bytes     receive Timestamp (ms since 0:00 GMT)   8523283
     4 bytes     transmit Timestamp (ms since 0:00 GMT)  8523283
```

Figure 29: Timestamp/Timestamp Reply message.

6.4.1.2.8 Information Request or Reply

Type (15, 16)	Code (0)	Checksum
Identifier		Sequence Number

Figure 30: Information Request/Reply message format.

The use of this message enables a host to determine the number of the network it is on. This message type is obsolete and should not be used (Comer, 1991). RARP and BOOTP can be used to perform this function.

6.4.1.2.9 Address Mask Request and Address Mask Reply

Type	Code (0)	Checksum
Identifier		Sequence Number
Address Mask		

Figure 31: Address Mask Request/Reply message format.

This message type is used to determine the address mask used to subnet a network. Networks can be subnetted according to the process outlined in RFC 950, to increase their performance and logical organisation

6.4.1.3 Implementation

ICMP is an uncomplicated protocol, and therefore was quite simple to decode.

1) The type of the ICMP message was determined and used as the variable for a switch statement. The appropriate function was chosen from this statement.

```
register struct icmp *p = (struct icmp *)malloc(sizeof(struct icmp));

switch (p->icmp_type) {
case ICMP_ECHOREPLY:
     Decode_ICMP_ECHOREPLY(p, len);
     break;
case ICMP_UNREACH:
     Decode_ICMP_UNREACH(p, len);
     break;
case ICMP_SOURCEQUENCH:
     Decode_ICMP_SOURCEQUENCH(p, len);
     break;
…    …    …
…    …    …
…    …    …
case ICMP_MASKREQ:
     Decode_ICMP_MASKREQ(p, len);
     break;
case ICMP_MASKREPLY:
     Decode_ICMP_MASKREPLY(p, len);
     break;
default:
     Decode_unknown_ICMP(p, len);
     break;
```

2) Each message type is decoded in a similar manner, as all messages have common components. For this reason, many macros were used to decode common arguments. *Ntohs(3N)* was used to convert values from network byte order into host byte order.

```
#defineHEADER(Text)  SetHeader(0, sprintf(lengthStr, "%u bytes", len), (Text))

#defineTYPE(Type)    sprintf(dataStr, "%d", p->icmp_type); \
                     SetFields(1, "1 byte", "type of message", dataStr, (Type))

#defineCODE(Code)    sprintf(dataStr, "%d", p->icmp_code); \
                     SetFields(1, "1 byte", "type sub code", dataStr, (Code))

#defineCHECKSUM      sprintf(dataStr, "0x%x", ntohs(p->icmp_cksum)); \
                     SetFields(1, "2 bytes", "checksum", dataStr, "")

#defineIDENTIFIER    sprintf(dataStr, "0x%x", ntohs(p->icmp_id)); \
                     SetFields(1, "2 bytes", "identifier", dataStr, "")

#defineSEQUENCE      sprintf(dataStr, "0x%x", ntohs(p->icmp_seq)); \
                     SetFields(1, "2 bytes", "sequence Number", dataStr, "")

#defineOPTIONAL      DataStr(dataStr, (u_char *)p->icmp_data, len-8); \
                     SetFields(1, sprintf(lengthStr, "%d bytes", len-8), \
                          "optional Data", dataStr, "")

static void Decode_ICMP_ECHO(p, len)
register struct icmp *p;
int len;
{
HEADER("ICMP Packet - Echo or Echo Reply Message");

TYPE("Echo Message");
```

```
CODE("");

CHECKSUM;

IDENTIFIER;

SEQUENCE;

OPTIONAL;
}
```

6.4.2 Remote Procedure Calls (RPC)

6.4.2.1 Overview

Sun RPC (Sun, 1988) provides for easily implemented distributed systems, using an approach analogous to program procedure calls. The RPC specification defines two end points for communications: a server and a client, which are normally running on different machines an arbitrary distance apart. To use a procedure provided by a server, the client will send a message, using UDP or TCP, to that server.

This message will identify the program to be executed (e.g. NFS), the version of it to be executed and also the procedure number required. The server will execute the chose procedure with the arguments supplied by the message, and return any results to the client upon completion. Figure 32 illustrates this operation.

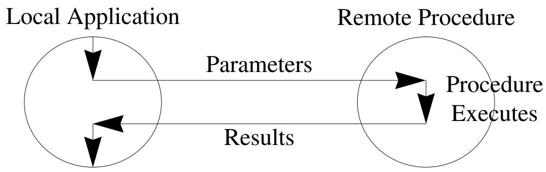


Figure 32: RPC operation (Corbin, 1991).

RPC programs (servers) can be started or stopped at any time, and so a register must be kept on each machine. This register must keep track of all programs available, the services (UDP or TCP) they support, and the ports on which they can be contacted. This service is provided by *portmapper*, which is an RPC application defined in RFC 1057. Portmapper is described in section 6.4.5.

RPC depends on two closely related components: XDR and rpcgen.

6.4.2.2 eXternal Data Representation (XDR)

XDR, defined in RFC 1014 (Sun, 1987), is a standard for machine independent data representation. Such representation (which is also provided by ASN.1) is necessary as different architectures store data in different ways (e.g. The big endian/little endian problem). Data in XDR format can be transported between different architectures at will.

XDR also provides a library of functions that can be used to encode (decode) data to (from) the XDR formats. This library consists of two types of functions (Corbin, 1991): Those which create and manipulate XDR streams, and those which convert data to or from these streams. The latter functions are referred to as filters. XDR filters can be used for both encoding and decoding — the operation which they perform on the stream depends on the “mode” the stream is in. Filters are usually created from XDR data definitions using rpcgen.

6.4.2.3 Rpcgen

The XDR specification also describes a language which can be used to describe data types. This is a C-like language used only for data type definitions — it is not a programming language. The use of this language allows the description of arbitrary data structures in a concise manner. Figure 33 shows the definition of the settable attributes structure from NFS.

```
/*
 * File attributes which can be set
 */
struct sattr {
     unsigned mode; /* protection mode bits */
     unsigned uid; /* owner user id */
     unsigned gid; /* owner group id */
     unsigned size; /* file size in bytes */
     nfstime atime; /* time of last access */
     nfstime mtime; /* time of last modification */
};
```

Figure 33: NFS settable attributes definition.

When used in conjunction with the RPC programming language defined in RFC 1057, the XDR language can be used to describe RPC programs. RPC itself is described with the XDR language, while NFS is described with both languages. RPC program descriptions can be compiled with *rpcgen*, which produces several different modules.

These modules include the client and server stubs used as a basis for the implementation of the client and server, a header file with definitions as well as the XDR filter routines for all data types defined. The use of rpcgen greatly reduces the effort needed to implement RPC programs, and was used in this project to create the XDR filters for most RPC programs decoded.

Figure 34 shows the filter used for the structure shown in figure 33, as created by rpcgen. This filter is constructed of other filters. When faced with choices (switches) XDR filters will determine from the data which choice is valid and follow that branch of the structure. This enables complex structures to be encoded or decoded with one call.

```
bool_t
xdr_sattr(xdrs, objp)
     register XDR *xdrs;
     sattr *objp;
{
     register long *buf;

     if (!xdr_u_int(xdrs, &objp->mode))
          return (FALSE);
     if (!xdr_u_int(xdrs, &objp->uid))
          return (FALSE);
     if (!xdr_u_int(xdrs, &objp->gid))
          return (FALSE);
     if (!xdr_u_int(xdrs, &objp->size))
          return (FALSE);
```

```
        if (!xdr_nfstime(xdrs, &objp->atime))
            return (FALSE);
        if (!xdr_nfstime(xdrs, &objp->mtime))
            return (FALSE);
        return (TRUE);
}
```

Figure 34: NFS settable attributes XDR filter.

6.4.2.4 Message Structure

6.4.2.4.1 Call Messages

In addition to the procedure arguments, RPC call messages contain information necessary to distinguish the RPC program and procedure being called. Figure 35 shows the structure of a call message.

XID
Message Type
RPC Version
Program Number
Program Version
Procedure Number
Authentication Parameters
Authentication Verifier
Arguments

Figure 35: RPC call message format.

The `XID` is a unique identifier used to match reply messages with the corresponding call message. The `message type` is 0 for a call message and 1 for a reply message. `RPC version` indicates the version of RPC being used and must be equal to two. The `program number` identifies the program being called. Program numbers of well used RPCs are assigned by Sun. The `version number` is used to distinguish the version of the program to run. This allows multiple versions of a program to be supported simultaneously, and allows for backward compatibility. The `procedure number` specifies the procedure number to be called.

The `authentication credentials` identify the client to the server. The various types of authentication are described in section 6.4.2.4.3. The `authentication verifier` is used to verify the validity of the credentials. i.e. To make sure the client is who it says it is.

Finally, the `arguments` are procedure specific and are not interpreted by RPC.

6.4.2.4.2 Reply Messages

RPC reply messages have a structure quite different to that of the call messages and don't contain any data identifying the program the message belongs to.

XID
Message Type
Reply Status
Authentication Verifier
Accept Status
Results

Figure 36: RPC reply message format.

The `XID` and `message type` are as for those in the call message. The lack of identifying information in the message means the client must store information about all calls it has made, so that replies can be matched. The `reply status` indicates the success of the RPC call. It gives no indication as to the success of the RPC procedure, however. An RPC call can be successful but fail for other reasons e.g. Bad credentials or wrong RPC version. If the RPC call is unsuccessful, no more data follows.

The `authentication verifier` verifies the server to the client. The `accept status` indicates the result of the RPC. This indicates whether the procedure executed successfully or if it failed. A procedure can fail for many reasons. e.g. program mismatch, invalid procedure number and bad arguments. If the procedure executed successfully then results will follow.

The `results` usually indicate the outcome of a procedure's execution.

6.4.2.4.3 Authentication

Authentication is used to identify clients and servers to one another, and also to check the validity of this identity. In addition to the authentication "flavours" defined by users, four flavours are defined by Sun in RFC 1057.

6.4.2.4.3.1 Null Authentication (AUTH_NULL)

When a server does not care who uses it, or the client does not know who it is, AUTH_NULL can be used. This type of authentication offers no security.

6.4.2.4.3.2 Unix Authentication (AUTH_UNIX)

This flavour is used when the client wishes to identify itself as it is identified in a Unix environment. i.e. By using its user name, group names, machine name etc. AUTH_UNIX has no verifier, so a clever user can easily fake credentials and gain access to another user's data. The verifier field in the reply message may use AUTH_NULL or AUTH_SHORT.

6.4.2.4.3.3 Short Hand Unix Authentication (AUTH_SHORT)

When AUTH_SHORT is returned in the verify field of a reply message, it gives the caller the option of using it in future RPC calls. As typical AUTH_UNIX structures can be quite large, it gives the client the opportunity to reduce both bandwidth and CPU usage.

6.4.2.4.3.4 DES Authentication (AUTH_DES)

AUTH_DES can be used to provide secure RPC, and also to overcome AUTH_UNIX's unix orientated naming style. The naming problem is overcome by identifying the caller with a unique and system specific string of characters. The security is provided by using the authentication verifier field in the call and reply messages. The verify consists mainly of an encrypted timestamp, which the server can decrypt and check the validity of. This timestamp can only be valid if the client is the correct one.

6.4.2.5 Implementation

6.4.2.5.1 XDR

As mentioned earlier, the XDR library contains filters and functions used to manipulate XDR streams. An XDR stream is simply an area in memory to (from) which data is encoded (decoded). An XDR stream can be created with the following code:

```
#define ETHER_MTU 1500
#define XDR_BUFFSIZE ETHER_MTU

char xdr_buff[XDR_BUFFSIZE];
XDR xdrs;

xdrmem_create(&xdrs, xdr_buff, XDR_BUFFSIZE, XDR_DECODE);
```

`XDR_DECODE` causes this stream to be initialised to decode data from the buffer `xdr_buff`. To initialise the stream for encoding, it is simply a case of replacing the `XDR_DECODE` with `XDR_ENCODE`.

A filter is passed a pointer to the stream, as well as a pointer to the data type it encodes/decodes. If the stream is `XDR_DECODE` then data will be decoded from the stream's buffer and placed in the structure. A value of `XDR_ENCODE` causes data to be encoded from the structure into the stream's buffer.

6.4.2.5.2 Matching Call/Reply Messages

As RPC reply messages don't contain program/procedure information, it is necessary to match them with their call messages. This is performed by storing for each call message the `XID`, `program number`, `program version` and `procedure number`. Several methods of storage were considered for this data.

6.4.2.5.2.1 Arrays

This was the simplest method, which also incurred much overhead. Once the data was placed in the array, the array would need to be sorted before a binary search could be used to retrieve a required entry. This method was used to test whether matching worked, but was discarded afterwards.

6.4.2.5.2.2 Height Balanced (AVL) Trees

Originally, it was thought that AVL trees would be the ideal solution for the storage required. Nodes in AVL trees are dynamically allocated, and can be inserted into the tree at the correct position. This means that the tree need never be sorted, although it would need to be rebalanced to keep its efficiency.

Code for implementation of an AVL tree (Appleton, 1989) was obtained and was linked in with the source code. When difficulty arose with the implementation of the tree, it was decided to abandon it and use a different approach.

6.4.2.5.2.3 Hashtables

Packetman already used hashtables for the ethernet address/machine name resolution. It was a simple case to use these for the storage of the data. With the `XID` used as the hash value, only the `program number`, `program version` and the `procedure number` need be stored. The only alteration to the source code required to make this operational was to add a function to allocate memory for the data stored:

```
struct call_data {
      u_long prog;
      u_long vers;
      u_long proc;
};

static struct call_data *
Alloc_call_data(cd)
struct call_data *cd;
{
      struct call_data *new;

      if((new = (struct call_data *)malloc(sizeof(struct call_data))) != NULL) {
            memcpy(new, cd, sizeof(struct call_data));
      }

      return(new);
}
```

This was the preferred solution as it used functions already available and involved the minimum change to the code.

6.4.2.5.3 Decoding

6.4.2.5.3.1 Overview

RPC programs are identified in the function `Decode_UDP`, which decodes the UDP header and determines the function used to decode the UDP data. The RPC program is determined, and then the function to decode this program is called. This function then calls the RPC routines to decode the RPC header and continues on to decode the protocol when they return.

It was decided to use this methodology rather than calling the RPC routines from `Decode_UDP` (and letting them identify and dispatch to the program decoder) as it follows way all other protocol types are dispatched. This keeps the methodology constant throughout all protocols.

Each RPC program decoder has a table which holds information about that program. This table holds the program name, the procedure name/number mappings (for all versions of the program), as well as the XDR filters used to encode/decode the arguments and results. This information is passed to the RPC routines, which use it to identify the program and procedure names, as well as whether that version of the program can be decoded.

6.4.2.5.3.2 Call Messages

The decomposition of call messages is methodical. The values for the various fields are identified an output to the `StringList` structure describe in section 3.2.2.

1) An `rpc_msg` structure is allocated and zeroed, ready to store the data retrieved from the stream.

```
struct rpc_msg *p;

if ((p=(struct rpc_msg *)malloc(sizeof(struct rpc_msg))) == NULL) {
     return(NULL);
}

bzero(p, sizeof(struct rpc_msg));
```

2) The addresses of the memory area set aside for the credentials is loaded into the structure, as this is where the authentication data is placed by `xdr_callmsg`.

```
char cred_area[2*MAX_AUTH_BYTES];

bzero(cred_area, 2*MAX_AUTH_BYTES);
p->rm_call.cb_cred.oa_base = cred_area;
p->rm_call.cb_verf.oa_base = &(cred_area[MAX_AUTH_BYTES]);
```

3) The packet is copied into the stream’s buffer. The position of the buffer is set to its start and the data is taken from the buffer using `xdr_callmsg`.

```
bcopy(packet, xdr_buff, len);
XDR_SETPOS(&xdrs, 0);

if (!xdr_callmsg(&xdrs, p)) {
```

```
        free(p);
        return(NULL);
    }
```

4) The name of the procedure is determined from the data and the information passed in. The length of the RPC header is also calculated.

```
    name = proc_name(p->rm_call.cb_proc, p->rm_call.cb_vers, program->procs);
    header_len = 40 + p->rm_call.cb_cred.oa_length + p->rm_call.cb_verf.oa_length;
```

5) The values for the various fields are added to the string structure described in section 3.2.2. The passed in data is used to identify the program and procedure names.

```
    #defineHEADER(Text)   SetHeader(0, sprintf(lengthStr, "%u bytes", header_len), \
                                (Text))

    #defineMESG_ID        sprintf(dataStr, "0x%x", p->rm_xid); \
                          SetFields(1, "4 bytes", "message ID", dataStr, "")

    #defineMESG_TYPE(x)   sprintf(dataStr, "%d", p->rm_direction); \
                          SetFields(1, "4 bytes", "message type", dataStr, (x))

    HEADER("Remote Procedure Call Header");

    MESG_ID;

    MESG_TYPE("Call");

    sprintf(dataStr, "%ld", p->rm_call.cb_rpcvers);
    SetFields(1, "4 bytes", "RPC version", dataStr, "");

    sprintf(dataStr, "%ld", p->rm_call.cb_prog);
    SetFields(1, "4 bytes", "RPC program", dataStr, program->name);

    sprintf(dataStr, "%ld", p->rm_call.cb_vers);
    /* make sure we know how to decode this version of the program before we try */

    if (p->rm_call.cb_vers <= program->size && p->rm_call.cb_vers > 0) {
        SetFields(1, "4 bytes", "program version", dataStr, "");
    } else {
        SetFields(1, "4 bytes", "program version", dataStr, "Unknown Program
            Version");
        free(p);
        return(NULL);
    }

    sprintf(dataStr, "%d", p->rm_call.cb_proc);
    SetFields(1, "4 bytes", "procedure number", dataStr, name);
```

6) The authentication fields are decoded. Currently, only AUTH_UNIX can be broken down into its constituent parts, as no other authentication types are used in the School of Computing. Other authentication flavours are represented as character strings. The data placed in `cred_area` is still in XDR format, as it is opaque[1]. This data is copied back to the start of the XDR buffer and decoded into the appropriate structure, and the fields are then output.

```
    struct opaque_auth *p;
```

```
struct authunix_parms auth;

pos = XDR_GETPOS(&xdrs); /* bad but easy */
bcopy(p->oa_base, xdr_buff, p->oa_length);
XDR_SETPOS(&xdrs, 0);

bzero(&auth, sizeof(struct authunix_parms));
xdr_authunix_parms(&xdrs, &auth);

XDR_SETPOS(&xdrs, pos);
```

7) Finally, the header identifying the arguments is printed. The function then returns and the caller takes over to decode the arguments.

```
sprintf(dataStr, "%s Arguments: %s", program->name, name);
SetHeader(0, sprintf(lengthStr, "%d bytes", len-header_len), dataStr);

return(p);
```

Figure 37 shows the header of a typical RPC call message.

```
96 bytes    Remote Procedure Call Header
     4 bytes     message ID                            0x4a220
     4 bytes     message type                          0                     Call
     4 bytes     RPC version                           2
     4 bytes     RPC program                           100000                Port Mapper
     4 bytes     program version                       2
     4 bytes     procedure number                      3                     Get Program Mapping
     64 bytes    Authentication Parameters
          4 bytes     authentication flavour           1                     Unix style
          4 bytes     stamp                            752309626             Wed Nov  3 14:53:46 1993
          5 bytes     machine name                     "marsh"
          4 bytes     user id                          0                     root
          4 bytes     group id                         0                     sys
          4 bytes     size of group array              7
          28 bytes    groups containing user           0                     sys
                                                       1                     daemon
                                                       2                     bin
                                                       3                     adm
                                                       4                     mail
                                                       42                    laser
                                                       62                    office
          4 bytes     length of data                   56
     8 bytes     Authentication Verifier
          4 bytes     authentication flavour           0                     No Authentication
          0 bytes     opaque data                      ""
          4 bytes     length of data                   0

16 bytes    Port Mapper Arguments: Get Program Mapping
     4 bytes     program number                        100005                Mount daemon
```

Figure 37: RPC call header.

6.4.2.5.3.3 Reply Messages

Decoding reply messages is more complex than call messages, as there are several forms which the message can take, depending upon the values of the reply status and accept status fields.

1) The rpc_msg structure is allocated, and the data is decoded from the XDR stream in a similar way to that of call messages. The major difference is that of setting the pointers to `xdr_void` and `NULL`. `Xdr_void` is a filter which does nothing, and is required as `xdr_replymsg` attempts to decode the results of the procedure. Setting these two values prevents this from hap-

1. Opaque data is that which is passed through an XDR stream uninterpreted (Corbin, 1991). When decoding from the stream, the data is placed straight into the allocated storage area without being altered. Authentication in RPC is always opaque, so the data must be decoded before use.

pening.

```
struct rpc_msg *p;
char cred_area[MAX_AUTH_BYTES];

if ((p=(struct rpc_msg *)malloc(sizeof(struct rpc_msg))) == NULL) {
     return(NULL);
}

bzero(p, sizeof(struct rpc_msg));
bzero(cred_area, MAX_AUTH_BYTES);

p->rm_reply.rp_acpt.ar_verf.oa_base = cred_area;
p->acpted_rply.ar_results.where = NULL;
p->acpted_rply.ar_results.proc = xdr_void;

bcopy(packet, xdr_buff, len);
XDR_SETPOS(&xdrs, 0);

if (!xdr_replymsg(&xdrs, p)) {
     free(p);
     return(NULL);
}
```

2) The length of the RPC header is found. This is involved, as to get the correct length all switches must be checked to determine the format of the message.

```
header_len = sizeof(p->rm_xid)+sizeof(p->rm_direction)+sizeof(p->
rm_reply.rp_stat);
switch(p->rm_reply.rp_stat) {
case MSG_ACCEPTED:
     header_len+=sizeof(p->rm_reply.rp_acpt.ar_verf.oa_flavor) +
          sizeof(p->rm_reply.rp_acpt.ar_verf.oa_length) +
          p->rm_reply.rp_acpt.ar_verf.oa_length +
          sizeof(p->rm_reply.rp_acpt.ar_stat);
     switch(p->rm_reply.rp_acpt.ar_stat) {
     case PROG_MISMATCH:
          header_len+=sizeof(p->rm_reply.rp_acpt.ar_vers);
          break;
     /* case SUCCESS: */
     /* case PROG_MISMATCH: */
     /* case PROG_UNAVAIL: */
     /* case PROC_UNAVAIL: */
     /* case GARBAGE_ARGS: */
     /* case SYSTEM_ERR: */
     default:
          break;
     }
     break;
case MSG_DENIED:
     header_len+=sizeof(p->rm_reply.rp_rjct.rj_stat);
     switch(p->rm_reply.rp_rjct.rj_stat) {
     case RPC_MISMATCH:
          header_len+=sizeof(p->rm_reply.rp_rjct.rj_vers);
          break;
     case AUTH_ERROR:
          header_len+=sizeof(p->rm_reply.rp_rjct.rj_why);
          break;
```

```
        default:
            break;
        }
        break;
    default:
        break;
    }
```

3) The matching call packet is then found.

```
    match = get_match(p->rm_xid);
```

4) The fields of the message are then decoded, again taking into account the values of the switches. If the RPC execution was a success, the header identifying the results is printed. The `rpc_msg` structure is then filled with the `program version` and `procedure number`, so that the caller can decode the results.

```
    HEADER("Remote Procedure Call Header");

    MESG_ID;

    MESG_TYPE("Reply");

    sprintf(dataStr, "%d" ,p->rm_reply.rp_stat);
    switch(p->rm_reply.rp_stat) {
    case MSG_ACCEPTED:
        SetFields(1, "4 bytes", "reply status", dataStr, "Message accepted");

        SetHeader(1, sprintf(lengthStr, "%d bytes", 8+p->
            rm_reply.rp_acpt.ar_verf.oa_length), "Authentication Verifier");
        Decode_auth(&p->rm_reply.rp_acpt.ar_verf);

        sprintf(dataStr, "%d", p->rm_reply.rp_acpt.ar_stat);
        switch(p->rm_reply.rp_acpt.ar_stat) {
        case SUCCESS:
            SetFields(1, "4 bytes", "accept status", dataStr, "Success");
            SetBlank();

            /* caller should decode this program specific data */
            if (match == NULL) {
                name = "Could not find matching call message";
                free(p);
                p = NULL;
            } else {
                name = proc_name(match->proc, match->vers, program->procs);
                p->rm_call.cb_vers = match->vers;
                p->rm_call.cb_proc = match->proc;
            }

            sprintf(dataStr, "%s Results: %s", program->name, name);
            SetHeader(0, sprintf(lengthStr, "%d bytes", len-header_len), dataStr);

            return(p);
            break;
        case PROG_MISMATCH:
            SetFields(1, "4 bytes", "accept status", dataStr, "Program mismatch");
```

```
            sprintf(dataStr, "%d", p->rm_reply.rp_acpt.ar_vers.low);
            SetFields(1, "4 bytes", "lowest program version", dataStr, "");

            sprintf(dataStr, "%d", p->rm_reply.rp_acpt.ar_vers.high);
            SetFields(1, "4 bytes", "highest program version", dataStr, "");
            break;
        case PROG_UNAVAIL:
            SetFields(1, "4 bytes", "accept status", dataStr,
                "Program unavailable");
            break;
        case PROC_UNAVAIL:
            SetFields(1, "4 bytes", "accept status", dataStr,
                "Procedure unavailable");
            break;
        case GARBAGE_ARGS:
            SetFields(1, "4 bytes", "accept status", dataStr, "Garbage arguments");
            break;
        case SYSTEM_ERR:
            SetFields(1, "4 bytes", "accept status", dataStr,
                "Some unknown error");
            break;
        default:
            SetFields(1, "4 bytes", "accept status", dataStr,
                "Unkown reject status");
            break;
        }
        break;
    case MSG_DENIED:
        SetFields(1, "4 bytes", "reply status", dataStr, "Message denied");

        sprintf(dataStr, "%d", p->rm_reply.rp_rjct.rj_stat);
        switch(p->rm_reply.rp_rjct.rj_stat) {
        case RPC_MISMATCH:
            SetFields(1, "4 bytes", "reject status", dataStr,
                "RPC version mismatch");

            sprintf(dataStr, "%d", p->rm_reply.rp_rjct.rj_vers.low);
            SetFields(1, "4 bytes", "lowest RPC version", dataStr, "");

            sprintf(dataStr, "%d", p->rm_reply.rp_rjct.rj_vers.high);
            SetFields(1, "4 bytes", "highest RPC version", dataStr, "");

            break;
        case AUTH_ERROR:
            SetFields(1, "4 bytes", "reject status", dataStr,
                "Authentication error");
            Decode_AUTH_ERROR(p->rm_reply.rp_rjct.rj_why);
            break;
        default:
            SetFields(1, "4 bytes", "reject status", dataStr,
                "Unkown reject status");
            break;
        }
        break;
    default:
        SetFields(1, "4 bytes", "reply status", dataStr, "Unknown reply");
        break;
    }
```

```
SetBlank();

return(NULL);
```

Figure 38 shows a typical RPC reply message.

```
24 bytes    Remote Procedure Call Header
     4 bytes     message ID                          0x4a220
     4 bytes     message type                        1               Reply
     4 bytes     reply status                        0               Message accepted
     8 bytes     Authentication Verifier
          4 bytes     authentication flavour         0               No Authentication
          0 bytes     opaque data                    ""
          4 bytes     length of data                 0
     4 bytes     accept status                       0               Success

4 bytes     Port Mapper Results: Get Program Mapping
     4 bytes     program port number                 1051
```

Figure 38: RPC reply header.

6.4.2.5.3.4 Half Decoding

As well as creating the protocol stack, the information for the call packets is stored into the hash-table when this function is used. These operations are performed as follows:

1) The direction of the packet is determined.

```
if (!get_dir(&xid, &dir)) {
      return;
}
```

2) For call messages, an `rpc_msg` structure is allocated, the packet copied into the XDR buffer, the credentials area setup and the data is decoded from the stream.

```
struct rpc_msg *p;
struct call_data cd;
char cred_area[2*MAX_AUTH_BYTES];

switch(dir) {
case CALL:
      if ((p = (struct rpc_msg *)malloc(sizeof(struct rpc_msg))) == NULL) {
            return;
      }

      bzero(xdr_buff, XDR_BUFFSIZE);
      bcopy(packet, xdr_buff, len);
      bzero(p, sizeof(struct rpc_msg));
      bzero(cred_area, 2*MAX_AUTH_BYTES);

      XDR_SETPOS(&xdrs, 0);

      /* set up the base area for the authentication */
      p->rm_call.cb_cred.oa_base = cred_area;
      p->rm_call.cb_verf.oa_base = &(cred_area[MAX_AUTH_BYTES]);

      /* get the call message */
      if (!xdr_callmsg(&xdrs, p)) {
            free(p);
            return;
      }
```

3) The call message details are then added to the hashtable.

```
cd.prog = p->rm_call.cb_prog;
cd.vers = p->rm_call.cb_vers;
cd.proc = p->rm_call.cb_proc;

/* add the packet details to the table */
sprintf(tmp, "%ld", p->rm_xid);
SetHashTable(call_packets, tmp, &cd);
```

4) The protocol stack is then created.

```
sprintf(dataStr, "/call/xid:0x%lx,proc:%ld", p->rm_xid, p->rm_call.cb_proc);
strcat(protocolStack, dataStr);
```

5) In the case of reply messages, only the operations on the protocol stack are performed.

```
case REPLY:
     sprintf(dataStr, "/reply/xid:0x%lx", xid);
     strcat(protocolStack, dataStr);
```

Figure 39 shows the protocol stack of the RPC messages in figures 37 and 38.

```
1  01/01/70 10:22:16.877550   156  08:00:69:06:39:ba     marsh -> lillee    08:00:69:06:3a:25  Ether/IP/UDP/sunrpc/call/xid:0x4a220,proc:3
2  01/01/70 10:22:16.881450    72  08:00:69:06:3a:25    lillee -> marsh     08:00:69:06:39:ba  Ether/IP/UDP/sunrpc/reply/xid:0x4a220
```

Figure 39: RPC protocol stack.

6.4.3 Network File System (NFS)

6.4.3.1 Overview

NFS (Sun, 1989) is an RPC program used for the sharing of file systems across heterogeneous architectures. The use of NFS offers many benefits to the user, some of which are:

- Much disk space is saved as commonly used files can be installed on only one machine, and then used via NFS by others. This frees up space for other purposes. The `/usr/local` hierarchy is often used in this manner.

- Administrative effort is reduced by the centralisation of files NFS affords. For example, if `/usr/local` is NFS mounted to many machines, changes to its contents (such as upgrading the libraries) need only be performed in one place to effect all machines using the file system.

- Common disk areas can be made available to many machines. This commonly occurs with user areas, which must be available to all machines on which a user has an account. It is clearly unacceptable for the user to be given separate areas on each machine.

A file system exported by an NFS server can be mounted by an NFS client to make it available to users on that machine. Once mounted, the file system appears to the user as if it is local to the machine — the user need not know the system is mounted remotely.

When accessing files, the operating system will determine whether the file is local or remote. In the latter case, the operation will be passed to the NFS client on that machine. It will send an RPC message to the appropriate server, which will execute the and then return the results back to the client.

6.4.3.2 NFS Procedures

The procedures for NFS and the other RPC programs decoded during the project will be explained with a picture of the arguments and results (if one is available) as well a brief explanation of the data structures.

6.4.3.2.1 Null Procedure

The Null procedure, provided in all RPC programs, does no work. It can be used to test whether the server is responding, and if so what the time lag may be. The Null procedure has no arguments and returns no results.

The Null procedure will not be described in the other RPC programs mentioned in this report

6.4.3.2.2 Get File Attributes

Arguments:

```
32 bytes    Network File System Arguments: Get File Attributes
    32 bytes    file file-handle (opaque)              "............)............)..." . . .
```

Figure 40: Get File Attribute arguments.

The `file handle` passed to this procedure indicates the file whose attributes are required. NFS uses file handles rather that file names to reference files (except when first accessing a file). File handles are opaque (read machine specific) and so no attempt is made to decode them further. *Packetman* can't know what format the file handle is — it could be destined for any type of machine.

Results:

```
72 bytes    Network File System Results: Get File Attributes
    4 bytes     NFS reply status                   0                        No Error
    68 bytes    File Attributes
        4 bytes     file type                      2                        Directory
        4 bytes     file mode                      043777                   drwxrwsrwt
        4 bytes     number of hard links           4
        4 bytes     owner user id                  0                        root
        4 bytes     owner group id                 4                        mail
        4 bytes     file size (bytes)              11776
        4 bytes     preferred block size (bytes)   4096
        4 bytes     special device number          0
        4 bytes     number of blocks used by file  23
        4 bytes     device number                  5805
        4 bytes     inode number                   2
        8 bytes     time of last access            754820649s 0us           Thu Dec  2 16:24:09.0 1993
        8 bytes     time of last modification      754820565s 0us           Thu Dec  2 16:22:45.0 1993
        8 bytes     time of last change            754820565s 0us           Thu Dec  2 16:22:45.0 1993
```

Figure 41: Get File Attribute results.

The `reply status` indicates the outcome of the procedure execution. There are many possible values for this field, but in this case it indicates that the procedure executed correctly. The `file attributes` are the file attributes requested in the call. The `file type` indicates the type of the file (regular, directory, FIFO etc.). The `file mode` indicates the permissions for the file. The `number of hard links` is the number of names for the same file. The `owner user` and `group ids` are those of the owner of the file. The `file size` is in bytes. The `preferred block size` is the size of one block of the file. The `special device number` is used when the file is a block or character special type. The `number of blocks` used by the file is the number of `blocksize` blocks it uses. The `device number` is the number of the device on which the file is present. The `inode number` uniquely identifies the file in the file system. The three `time` fields indicate the access times for the file.

6.4.3.2.3 Set File Attributes

Arguments:

```
64 bytes    Network File System Arguments: Set File Attributes
    32 bytes    file file-handle (opaque)              "................@.........,..@" . . .
    32 bytes    Settable Attributes
        4 bytes     file mode                          04755                   -rwsr-xr-x
        4 bytes     owner user id                      -1                      Not set
        4 bytes     owner group id                     -1                      Not set
        4 bytes     file size (bytes)                  -1                      Not set
        8 bytes     time of last access                -1s -1us                Not set
        8 bytes     time of last modification          -1s -1us                Not set
```

Figure 42: Set File Attributes arguments.

The `file handle` indicates the file whose attributes are to be set. The `settable attributes` are those attributes of a file that can be change, and have the same function as those found in the file attributes structure. Values of `-1` indicate that that value is not set and should be ignored. The server will then set these attributes of the file as it sees it.

Results:

```
72 bytes    Network File System Results: Set File Attributes
    4 bytes     NFS reply status                   0                       No Error
    68 bytes    File Attributes
        4 bytes     file type                      1                       Regular
        4 bytes     file mode                      0104755                 -rwsr-xr-x
        4 bytes     number of hard links           1
        4 bytes     owner user id                  0                       root
        4 bytes     owner group id                 20                      project
```

Figure 43: Set File Attributes results.

The new file attributes are returned.

6.4.3.2.4 Get Filesystem Root

This procedure is obsolete. Its function is performed by the “Add Mount Entry” procedure in NFS Mount.

6.4.3.2.5 Look Up File Name

Arguments:

```
44 bytes    Network File System Arguments: Look Up File Name
    32 bytes    directory file-handle (opaque)         "...........%+../..........+..  " . . .
    8 bytes     filename                               "yppasswd"
```

Figure 44: Look Up File Name arguments.

The `file handle` indicates to the server the directory in which to look up the file name `filename`. This is one of the few cases were a file is not referenced by a file handle.

Results:

```
104 bytes   Network File System Results: Look Up File Name
    4 bytes     NFS reply status                        0                    No Error
    32 bytes    file file-handle (opaque)               "............+..T.........+.. " . . .
    68 bytes    File Attributes
        4 bytes     file type                           1                    Regular
        4 bytes     file mode                           0100600              -rw-------
        4 bytes     number of hard links                1
        4 bytes     owner user id                       2016                 pug
        4 bytes     owner group id                      20                   project
```

Figure 45: Look Up File Name results.

The procedure returns the `reply status`, the `file handle` to be used for future reference to the looked up file as well as the file's `file attributes`.

6.4.3.2.6 Read From Symbolic Link

Arguments:

```
32 bytes    Network File System Arguments: Read From Symbolic Link
    32 bytes    file file-handle (opaque)               "............+..<.........+.. " . . .
```

Figure 46: Read From Symbolic Link arguments.

The file handle is that of the symbolic link whose contents are to be read.

Results:

```
28 bytes    Network File System Results: Read From Symbolic Link
    4 bytes     NFS reply status                        0                    No Error
    24 bytes    Path
        4 bytes     length of string                    20
        20 bytes    contents of symbolic link           "units/project/report"
```

Figure 47: Read From Symbolic Link results.

The `path` is a counted array of characters which is the contents of the symbolic link. As symbolic links are always full path names or relative to the current directory, the client can use lookups to obtain a file handle for the file.

6.4.3.2.7 Read From File

Arguments:

```
44 bytes    Network File System Arguments: Read From File
    32 bytes    file file-handle (opaque)               "............). ..........). ." . . .
    4 bytes     byte offset in file                     122880
    4 bytes     immediate read count                    4096
    4 bytes     total read count (from offset)          4096
```

Figure 48: Read From File arguments.

The `file handle` indicates the file from which to read. The `byte offset` is the offset in the file from which to start the read. `Immediate read count` bytes are read. `Total read count` is unused.

Results:

```
76 bytes    Network File System Results: Read From File
    4 bytes     NFS reply status                        0                       No Error
    68 bytes    File Attributes
        4 bytes     file type                           1                       Regular
        4 bytes     file mode                           0100644                 -rw-r--r--
        4 bytes     number of hard links                1
        4 bytes     owner user id                       2010                    heskethm
        4 bytes     owner group id                      20                      project
        4 bytes     file size (bytes)                   117130
        4 bytes     preferred block size (bytes)        4096
        4 bytes     special device number               4294967295
        4 bytes     number of blocks used by file       229
        4 bytes     device number                       5792
        4 bytes     inode number                        2843
        8 bytes     time of last access                 754825907s 0us          Thu Dec  2 17:51:47.0 1993
        8 bytes     time of last modification           754825314s 0us          Thu Dec  2 17:41:54.0 1993
        8 bytes     time of last change                 754825314s 0us          Thu Dec  2 17:41:54.0 1993
    4 bytes     size of data (bytes)                    0
    0 bytes     data                                    ""
```

Figure 49: Read From File results.

This procedure returns the `file attributes` of the file after the read, as well as the read data. The `size of data` indicates the number of bytes read, in this case none. The read procedure will attempt to read from the `offset` up to the `read count`, but in some cases may read less.

6.4.3.2.8 Write To Cache

This procedure is for use in the next version of the protocol.

6.4.3.2.9 Write To File

Arguments:

```
52 bytes    Network File System Arguments: Write To File
    32 bytes    file file-handle (opaque)               "..........g).!..........). ." . . .
    4 bytes     beginning offset in file                0                       Ignored
    4 bytes     current offset in file                  0
    4 bytes     total write count (to offset)           2                       Ignored
    4 bytes     size of write (bytes)                   2
    2 bytes     data                                    ".."
```

Figure 50: Write To File arguments.

The `file handle` indicates the file to which the write is to take place. The `beginning offset` and `total write count` are ignored. The `current offset` indicates the position in the file from which to commence the write. If it is equal to the file length then the data will be appended. The `size of write` is the number of bytes to write, and `data` is the data to be written.

Results:

```
72 bytes    Network File System Results: Write To File
    4 bytes     NFS reply status                        0                       No Error
    68 bytes    File Attributes
        4 bytes     file type                           1                       Regular
        4 bytes     file mode                           0100644                 -rw-r--r--
        4 bytes     number of hard links                1
        4 bytes     owner user id                       2010                    heskethm
        4 bytes     owner group id                      20                      project
```

Figure 51: Write To File results.

This procedure returns the status and the file attributes of the file after the write operation.

6.4.3.2.10 Create File

Arguments:

```
72 bytes    Network File System Arguments: Create File
    32 bytes    directory file-handle (opaque)          "...........7).!...........). ." . . .
    4 bytes     filename                                "core"
    32 bytes    Settable Attributes
        4 bytes     file mode                           0100644                -rw-r--r--
        4 bytes     owner user id                       -1                     Not set
        4 bytes     owner group id                      -1                     Not set
        4 bytes     file size (bytes)                   0
        8 bytes     time of last access                 -1s -1us               Not set
        8 bytes     time of last modification           -1s -1us               Not set
```

Figure 52: Create File arguments.

The `file handle` indicates the directory in which to create the file `filename`, with the initial attributes `settable attributes`. This procedure cannot be used to create directories.

Results:

```
104 bytes   Network File System Results: Create File
    4 bytes     NFS reply status                        0                      No Error
    32 bytes    file file-handle (opaque)               "............). ...........). ." . . .
    68 bytes    File Attributes
        4 bytes     file type                           1                      Regular
        4 bytes     file mode                           0100644                -rw-r--r--
        4 bytes     number of hard links                1
        4 bytes     owner user id                       2010                   heskethm
        4 bytes     owner group id                      20                     project
```

Figure 53: Create File results.

The results include the `status`, the `file handle` for the created file plus its initial `file attributes`.

6.4.3.2.11 Remove File

Arguments:

```
48 bytes    Network File System Arguments: Remove File
    32 bytes    directory file-handle (opaque)          "............(;g...........(;f." . . .
    10 bytes    filename                                ",perth3.pv"
```

Figure 54: Remove File arguments.

This procedure removes the file `filename` from the directory indicated by the file handle `file handle`. This procedure can't be used to remove directories.

Results:

```
4 bytes     Network File System Results: Remove File
    4 bytes     NFS reply status                        0                      No Error
```

Figure 55: Remove File results.

The `status` of the operation is returned.

6.4.3.2.12 Rename File

Arguments:

```
100 bytes   Network File System Arguments: Rename File
    32 bytes    directory file-handle (opaque)          "...............4P.Wb........h." . . .
    10 bytes    filename                                "homer_kroq"
    32 bytes    directory file-handle (opaque)          "...............4P.Wb........h." . . .
    14 bytes    filename                                "homer_kroq.wav"
```

Figure 56: Rename File arguments.

This procedure renames the file in the first directory/filename pair to the file in the second directory/filename pair.

Results:

```
4 bytes     Network File System Results: Rename File
    4 bytes     NFS reply status                       0                          No Error
```

Figure 57: Rename File results.

The `status` of the operation is returned.

6.4.3.2.13 Create Link To File

Arguments:

```
80 bytes    Network File System Arguments: Create Link To File
    32 bytes    old-file file-handle (opaque)           "..........Vt(;g...........(;f." . . .
    32 bytes    directory file-handle (opaque)          "............(;g...........(;f." . . .
    10 bytes    filename                                ",perth3.pv"
```

Figure 58: Create Link To File arguments.

The file `filename` is created in the directory `directory file handle`. The file created is a hard link to the file specified by the `old-file file handle`.

Results:

```
4 bytes     Network File System Results: Create Link To File
    4 bytes     NFS reply status                       0                          No Error
```

Figure 59: Create Link To File results.

The `status` of the operation is returned.

6.4.3.2.14 Create Symbolic Link

Arguments:

```
120 bytes   Network File System Arguments: Create Symbolic Link
    32 bytes    directory file-handle (opaque)          "............+..:..........+.. " . . .
    18 bytes    filename                                "staff-mail-archive"
    26 bytes    link name                               "/usr/data/marsh/mail/staff"
    32 bytes    Settable Attributes
        4 bytes     file mode                           0777                       -rwxrwxrwx
        4 bytes     owner user id                       -1                         Not set
        4 bytes     owner group id                      -1                         Not set
        4 bytes     file size (bytes)                   -1                         Not set
        8 bytes     time of last access                 -1s -1us                   Not set
        8 bytes     time of last modification           -1s -1us                   Not set
```

Figure 60: Create Symbolic Link arguments.

The symbolic link `filename` is created in the directory indicated by the `file handle`. The contents of the link (i.e. what it points to) is the path `link name`. The `settable attributes` of the arguments are never used on Unix servers.

Results:

```
4 bytes     Network File System Results: Create Symbolic Link
    4 bytes     NFS reply status                       0                          No Error
```

Figure 61: Create Symbolic Link results.

The `status` of the operation is returned.

6.4.3.2.15 Create Directory

Arguments:

```
104 bytes   Network File System Arguments: Create Directory
     32 bytes    directory file-handle (opaque)           "............+..:..........+.. " . . .
     34 bytes    filename                                 "supercalifragilisticexpialidoc" . . .
     32 bytes    Settable Attributes
          4 bytes     file mode                           0700                     -rwx------
          4 bytes     owner user id                       -1                       Not set
          4 bytes     owner group id                      20                       project
          4 bytes     file size (bytes)                   -1                       Not set
          8 bytes     time of last access                 -1s -1us                 Not set
          8 bytes     time of last modification           -1s -1us                 Not set
```

Figure 62: Create Directory arguments.

This procedure creates the directory indicated by `filename` in the directory indicated by the `file handle`. The initial attributes of the directory are those specified in the `settable attributes` structure.

Results:

```
104 bytes   Network File System Results: Create Directory
     4 bytes     NFS reply status                         0                        No Error
     32 bytes    file file-handle (opaque)                "...........;+..D.........+.. " . . .
     68 bytes    File Attributes
          4 bytes     file type                           2                        Directory
          4 bytes     file mode                           040700                   drwx------
          4 bytes     number of hard links                2
          4 bytes     owner user id                       2016                     pug
          4 bytes     owner group id                      20                       project
```

Figure 63: Create Directory results.

The results returned are as for those of the create file procedure.

6.4.3.2.16 Remove Directory

Arguments:

```
72 bytes    Network File System Arguments: Remove Directory
     32 bytes    directory file-handle (opaque)           "............+..:..........+.. " . . .
     34 bytes    filename                                 "supercalifragilisticexpialidoc" . . .
```

Figure 64: Remove Directory arguments.

The directory `filename` in the directory indicated by the `file handle` is removed. The directory indicated must be empty.

Results:

```
4 bytes     Network File System Results: Remove Directory
     4 bytes     NFS reply status                         0                        No Error
```

Figure 65: Remove Directory results.

The `status` of the operation is returned.

6.4.3.2.17 Read From Directory

Arguments:

```
40 bytes    Network File System Arguments: Read From Directory
     32 bytes    directory file-handle (opaque)           "...........%+../.........+.. " . . .
     4 bytes     offset in directory                      0
     4 bytes     size of read (bytes)                     4096
```

Figure 66: Read From Directory arguments.

The `file handle` indicates the directory from which the read is to take place. The `offset` is the position in the directory from which to start the read. The `size of read` is the maximum number of byte to read.

Results:

```
436 bytes   Network File System Results: Read From Directory
     4 bytes     NFS reply status                    0               No Error
     4 bytes     next offset                         18
     4 bytes     size of entries (bytes)             300
     4 bytes     last entry in result ?              0               False
     300 bytes   Directory Entries
          12 bytes    Directory Entry
               4 bytes     inode number              4133
               2 bytes     record length             12
               2 bytes     filename length           1
               1 bytes     filename                  "."
          12 bytes    Directory Entry
               4 bytes     inode number              728
               2 bytes     record length             12
               2 bytes     filename length           2
               2 bytes     filename                  ".."
          28 bytes    Directory Entry
               4 bytes     inode number              3430
               2 bytes     record length             28
               2 bytes     filename length           16
               16 bytes    filename                  "nfs-conversation"
          16 bytes    Directory Entry
               4 bytes     inode number              4057
               2 bytes     record length             16
```

Figure 67: Read From Directory results.

The `status` indicates the outcome of the read. The `next offset` is used in the next call to the procedure for further data from the directory. The `size of entries` is the number of bytes read. The `last entry in results` is false if there is more data available. Each directory entry contains the file's `inode number`, the `length` of the directory entry data structure, the `filename length` and the `filename` itself.

6.4.3.2.18 Get Filesystem Attributes

Arguments:

```
32 bytes   Network File System Arguments: Get Filesystem Attributes
     32 bytes    file-system file-handle (opaque)       "............+.. ...........+.. " . . .
```

Figure 68: Get Filesystem Attributes arguments.

The `file handle` indicates a file in the filesystem whose attributes are required.

Results:

```
24 bytes   Network File System Results: Get Filesystem Attributes
     4 bytes     NFS reply status                          0               No Error
     20 bytes    File System Attributes
          4 bytes     preferred transfer size (bytes)      8192
          4 bytes     filesystem block size (bytes)        512
          4 bytes     total blocks in file system          902658
          4 bytes     free blocks in file system           21920
          4 bytes     total blocks available to non-root   21920
```

Figure 69: Get Filesystem Attributes results.

The `filesystem attributes` contain the following data: The `preferred transfer size` is the optimal amount of data which the server prefers in read and write calls. The `block size` is the number of bytes in a block on the system. The `total blocks` is the number of block size blocks on the system. The `free blocks` is the number of free blocks. The `non-root` blocks are those available to non-privileged users.

6.4.3.3 Implementation

The decoding of all RPC programs is similar — the only difference is those functions used to decode the procedures. For this reason, the process of decoding RPC programs will only be discussed in this section.

6.4.3.3.1 Data Structures

As stated earlier, each RPC program decoded has a data structure containing various information about the program. The structures for NFS are shown below:

```
struct Procedures {
     u_char id;
     char *name;
     xdrproc_t rpc_xdrargs;
     xdrproc_t rpc_xdrres;
};

struct Procedures NFS2procedures[] = {
     {    RFS_NULL,"Null Procedure",xdr_void,xdr_void,},
     {    RFS_GETATTR,"Get File Attributes",xdr_fhandle,xdr_attrstat,},
     {    RFS_SETATTR,"Set File Attributes",xdr_saargs,xdr_attrstat,},
     {    RFS_ROOT,"Get Filesystem Root",xdr_void,xdr_void,},
     {    RFS_LOOKUP,"Look Up File Name",xdr_diropargs, xdr_diropres,},
     {    RFS_READLINK, "Read From Symbolic Link",xdr_fhandle,xdr_rdlnres,},
     {    RFS_READ,"Read From File",xdr_readargs, xdr_rdresult,},
     {    RFS_WRITECACHE, "Write To Cache",xdr_void,xdr_void,},
     {    RFS_WRITE,"Write To File",xdr_writeargs, xdr_attrstat,},
     {    RFS_CREATE,"Create File",xdr_creatargs, xdr_diropres,},
     {    RFS_REMOVE,"Remove File",xdr_diropargs, xdr_enum,},
     {    RFS_RENAME,"Rename File",xdr_rnmargs,xdr_enum,},
     {    RFS_LINK,"Create Link To File",xdr_linkargs, xdr_enum,},
     {    RFS_SYMLINK,"Create Symbolic Link",xdr_slargs,xdr_enum,},
     {    RFS_MKDIR,"Create Directory",xdr_creatargs, xdr_diropres,},
     {    RFS_RMDIR,"Remove Directory",xdr_diropargs, xdr_enum,},
     {    RFS_READDIR,"Read From Directory",xdr_rddirargs, xdr_getrddirres,},
     {    RFS_STATFS,"Get Filesystem Attributes",xdr_fhandle,xdr_statfs,},
};

int NFS2proceduresSize = (int)(sizeof(NFS2procedures) /
     sizeof(NFS2procedures[0]));

struct AllProcedures {
     struct Procedures *procedures;
     int size;
};

static struct AllProcedures NFSallProcedures[] = {
     {    NULL,0    },
     {    NULL,0    },
     {    NFS2procedures,RFS_NPROC},
};

struct Program {
     char *name;
     int size;
```

```
        struct AllProcedures *procs;
};

static struct Program NFSprogram = {
        {       "Network File System", (int)sizeof(NFSallProcedures)/
        sizeof(NFSallProcedures[0]), NFSallProcedures}
};
```

`NFS2procedures` contains an entry for each NFS procedure. For each procedure, the procedure number, name and the XDR filters for the arguments and results are stored. `NFSallProcedures` is used to store the procedure definitions for the various versions of the NFS protocol. At this time, only version two is supported. `NFSprogram` holds all structures for NFS. It includes the name of the program, the number of the highest version supported as well as all the procedures. This structure is passed to the RPC decoding routines.

6.4.3.3.2 Decoding RPC Headers

1) The direction of the packet is determined using the function `get_dir`, which is included with the RPC functions.

```
if (!get_dir(packet, &xid, &direction)) {
        return;
}
```

2) The call and reply headers are decoded in the same way. The RPC routines return a pointer to an `rpc_msg` structure, which is examined to discover the NFS version number. If this version can be decoded, the function will be called.

```
switch(direction) {
case CALL:
        if ((p=decode_CALL(packet, len, &NFSprogram, &xprt)) == NULL) {
                return;
        }
        switch(p->rm_call.cb_vers) {
        case 2:
                Decode_NFSv2(p, xprt);
                break;
        default:
                break;
        }
        free(p);
        break;
case REPLY:
        if ((p=decode_REPLY(packet, len, &NFSprogram, &xprt)) == NULL) {
                return;
        }
        switch(p->rm_call.cb_vers) {
        case 2:
                Decode_NFSv2(p, xprt);
                break;
        default:
                break;
        }
        free(p);
        break;
```

```
}
```

6.4.3.3.3 Decoding RPC Programs

From the `rpc_msg` structure passed to the program decoder, the direction of the message and the procedure number can be found. These can be used to determine which function to call. The operation for decoding call messages is shown, but those for reply messages are identical.

1) The direction of the message is checked, and the XDR filter used for the arguments/results is called to decode the arguments.

```
struct rpc_msg *p;
xdrproc_t func;

switch(p->rm_direction) {
case CALL:

     func = NFS2procedures[p->rm_call.cb_proc].rpc_xdrargs;
     (*func)(&xdrs, args);
```

2) The procedure arguments are decoded by calling the correct function. Fortunately, many procedures share arguments, and so the number of functions needed is less than the number of procedures available.

```
     switch(p->rm_call.cb_proc) {
     case RFS_NULL:
     case RFS_ROOT:
     case RFS_WRITECACHE:
          break;
     case RFS_GETATTR:
     case RFS_READLINK:
          Decode_fhandle_t((struct fhandle_t *)args, _FILE);
          break;
     …    …    …
     …    …    …
     …    …    …
     case RFS_STATFS:
          Decode_fhandle_t((struct fhandle_t *)args, FILE_SYSTEM);
          break;
     case RFS_SETATTR:
          Decode_sattrargs((struct sattrargs *)args);
          break;
     case RFS_READDIR:
          Decode_readdirargs((struct nfsrddirargs *)args);
          break;
     default:
          break;
     }
     break;
```

3) The code segment below shows how the arguments for the settable attributes procedure are decoded.

```
static void Decode_nfssattr(p)
register struct nfssattr *p;
```

```
{
      char name[40];

      SetHeader(1, "32 bytes", "Settable Attributes");

      MODE(p->sa_mode);

      sprintf(dataStr, "%ld", p->sa_uid);
      SetFields(1, "4 bytes", "owner user id", dataStr, Resolve_UID(p->sa_uid,
            name));

      sprintf(dataStr, "%ld", p->sa_gid);
      SetFields(1, "4 bytes", "owner group id", dataStr, Resolve_GID(p->sa_gid,
            name));

      SIZE(p->sa_size);

      AC_TIME(p->sa_atime);

      MOD_TIME(p->sa_mtime);
}

static void Decode_sattrargs(p)
register struct nfssaargs *p;
{
      Decode_fhandle_t(&p->saa_fh, "file ");

      Decode_nfssattr(&p->saa_sa);
}
```

6.4.3.3.4 Name Resolution

Many of the arguments and results in the NFS protocol return user id and group id values. It was therefore a natural progression on the decoding routines to resolve the numbers to the names.

Originally, *getpwuid(3V)* and *getgrgid(3V)* where used to retrieve this information. These functions caused unacceptable delays (up to two minutes), however, when the user id could not be found in the password file — *getpwuid* works by sequentially searching the password file. To overcome this problem, hash tables were created to hold the user id/username group id/groupname mappings.

At start up time, the id/name tuples are loaded into the hashtable from two files with the formats

```
root 0
daemon 1
bin 2
uucp 3
sys 4
adm 5
news 6
lp 9
nuucp 10
ftp 11
```

for the user names and

```
root 0
sys 0
daemon 1
bin 2
adm 3
mail 4
uucp 5
news 6
rje 8
lp 9
```

for the group names. These files are derivatives of the password and group files and are created with the commands “`ypcat passwd|cut -d: -f1,3|sed s/:/\ /g|sort -n +1 > filename`” or “`ypcat group|cut -d: -f1,3|sed s/:/\ /g|sort -n +1 > filename`”.

The use of these files caused the execution of the decoding routines to be significantly faster. If the hashtable does not exist at decoding time, *getpwuid* and *getgrgid* are used. The code for `Resolve_UID` is shown below. That for `Resolve_GID` is similar.

```
char *Resolve_UID(uid, name)
u_long uid;
char *name;
{
      struct passwd *password;
      char *tmp;

      if (uid == -1) {
            sprintf(name, "%s", NOT_SET);
            return(name);
      }

      if (passwdTable == NULL) {
            if ((password = getpwuid((uid_t)uid)) == NULL) {
                  sprintf(name, "Unknown User");
            } else {
                  strcpy(name, password->pw_name);
            }
      } else {
            sprintf(name, "%ld", uid);
            GetHashTable(passwdTable, name, &tmp);
            if (tmp == NULL) {
                  sprintf(name, "Unknown User");
            } else {
                  strcpy(name, tmp);
            }
      }
      return(name);
}
```

6.4.4 NFS Mount

6.4.4.1 Overview

The Mount protocol is related to, but separate from NFS (Sun, 1989). Mount provides operating specific services to initialise the NFS operations, and is used by the client to obtain the first file handle for the file system. This allows the initial access to this file system.

6.4.4.2 Mount Procedures

6.4.4.2.1 Add Mount Entry

Arguments:

```
20 bytes    Mount Protocol Arguments: Add Mount Entry
    16 bytes    directory path                          "/usr/marsh/staff"
```

Figure 70: Add Mount Entry arguments.

The `directory path` indicates the server the file system which the client wishes to mount.

Results:

```
36 bytes    Mount Protocol Results: Add Mount Entry
    4 bytes     result status                       0                          No Error
    32 bytes    file system file-handle (opaque)    "............)............)..." . . .
```

Figure 71: Add Mount Entry results.

The `status` indicates the outcome of the procedure. The `file handle` is the file handle that the client uses to access the file system just mounted.

6.4.4.2.2 Remove Mount Entry

Arguments:

```
12 bytes    Mount Protocol Arguments: Remove Mount Entry
    8 bytes     directory path                          "/usr/ftp"
```

Figure 72: Remove Mount Entry arguments.

The `directory path` indicates to the server the file system which the client wishes to unmount. This procedure returns no results.

6.4.4.2.3 Remove All Mount Entries

This procedure has no arguments or results. It causes the server to remove all mount list entries for the client.

6.4.4.2.4 Return Export List

This procedure causes the server to return to the client a list of all file systems it is willing to export, and to whom it will export them. No picture is available.

6.4.5 Portmapper (SunRPC)

6.4.5.1 Overview

The portmapper is used to map between RPC program numbers and transport specific port numbers (Sun, 1988). A portmapper server is present on each machine using RPC, and holds a table of all RPC servers available on that machine.

When initialising, RPC programs register themselves with the portmapper, and deregister when they become unavailable. Clients of the portmapper can use it to determine the port on which a particular RPC program is present. The portmapper also allows the use of broadcast RPC — a method by which the same RPC call can be sent to all servers on the network.

6.4.5.2 Portmapper Procedures

6.4.5.2.1 Register Program

This procedure is used when an RPC program initialise. It registers itself with the portmapper by sending its RPC program number, program version, the port on which it is resident and the service it supports (UDP or TCP). Calls to this procedure are made to the portmapper on the same machine as the RPC program, so they will never appear on the wire.

6.4.5.2.2 De-register Program

This procedure is used when a program wishes to deregister itself. It passes the same arguments as the register program procedure. Calls to this procedure will also not appear on the wire.

6.4.5.2.3 Get Program Mapping

Arguments:

```
16 bytes    Port Mapper Arguments: Get Program Mapping
     4 bytes     program number                    100004                Yellow Pages (NIS)
     4 bytes     program version                   2
     4 bytes     program protocol                  17                    UDP
     4 bytes     program port number               0
```

Figure 73: Get Program Mapping arguments.

The `program number` is the unique identifier for the RPC program whose port number is sought. The `version` is the version number sought. The `protocol` field indicates whether the program is using UDP or TCP for its communications. The `port number` field is unused.

Results:

```
4 bytes     Port Mapper Results: Get Program Mapping
     4 bytes     program port number                    715
```

Figure 74: Get Program Mapping results.

The `port number` which the program is using is returned. If the returned value is zero, it means that the program is not registered.

6.4.5.2.4 Get Program Mappings

This procedure is used to retrieve the mappings of all programs registered on the portmapper's machine. It takes no arguments and returns a list of mappings in the format shown in figure 73.

6.4.5.2.5 Call Remote Procedure

Arguments:

```
24 bytes    Port Mapper Arguments: Call Remote Procedure
    4 bytes     program number                      100004              Yellow Pages (NIS)
    4 bytes     program version                     1
    4 bytes     procedure number                    2
    4 bytes     arguments length                    8
    8 bytes     arguments                           "........"
```

Figure 75: Call Remote Procedure arguments.

The first three components of the arguments specify the RPC program to be called. `Arguments length` is the length of the arguments `arguments`. This procedure is used for broadcast RPC. As port numbers are allocated dynamically, the same RPC program may be present on different machines on different port numbers. To make it easier to contact all of the servers, broadcast RPC can be used.

The client broadcasts to port 111 (the portmapper) the program to be called and the arguments to be used. The portmapper on each machine on the network will then call the server on its machine with the arguments, and return the results to the client.

Results:

```
12 bytes    Port Mapper Results: Call Remote Procedure
    4 bytes     program port number                 1029
    4 bytes     results length                      4
    4 bytes     results                             "...."
```

Figure 76: Call Remote Procedure results.

`Program port number` specifies the port number of the called server on the machine which sent this reply message. The `results length` is the length of the results `results`. Figure 76 was one of many replies to the call.

7.0 Future Directions

7.1 BSD Packet Filter

The inefficiencies of NIT outlined in section 3.2.1.1.1 cause *packetman* to drop packets when used on a moderate to heavily loaded network. This loss of packets is important, as those missing may be crucial to discovering the cause of a network problem.

BPF could be used to overcome this problem, as it is up to 100 times more efficient than NIT. Unfortunately, BPF cannot be installed on many user's machines as it requires access to SunOS kernel source for installation. This source is available to few users. It is disappointing that Sun Microsystems has not made some accommodation so that their customers can install BPF. Perhaps this will be changed in the future.

McCanne and Jacobson (1993) describe BPF in detail.

7.2 File Selection Widget

A more complex file selection widget is needed to give more feedback to the user as to the contents of directories etc. Figure 77 shows the format of an ideal widget.

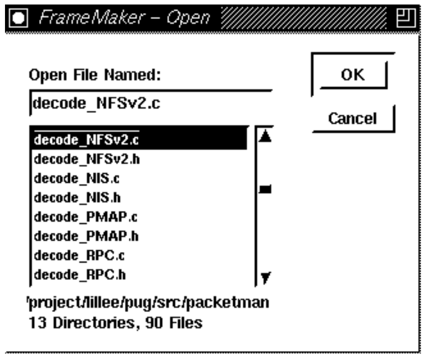


Figure 77: File Selection Widget (Frame Technology Corp., 1992)

7.3 Filtering

The area of *packetman* which requires most attention is the filter set and filter interface. The current menu driven filters are certainly easy to use, but lack the power of filter sets available in other tools. A new widget needs to be implemented to allow the selection and definition of filters. A possible format for this widget is shown below.

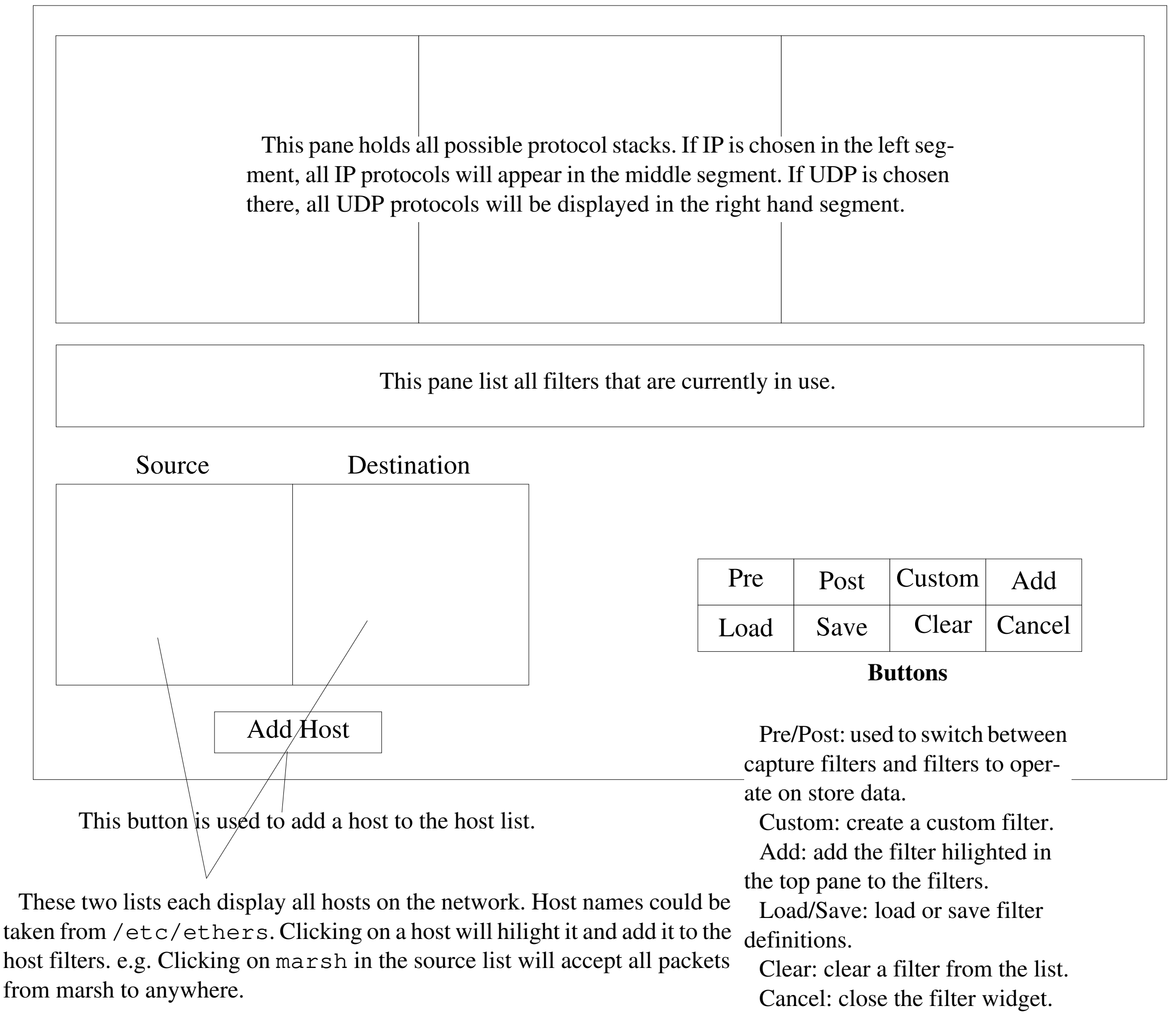


Figure 78: Filter Selection Widget.

7.4 Information Widget

This widget would be used to give information about *packetman*. Such information could include a copyright notice, a list of acknowledgements as well as the identification of authors. Many X11 applications have such widgets.

7.5 Options Widget

A widget allow the specification of options should be implemented. Some possible options would be: names of the support files, display colours, fonts, snap length, size of storage buffer, maximum number of packets etc. Once set these option could be stored in the user's home area, perhaps in `$HOME/.packetmanrc`, and would override any setting in the application resource file.

7.6 Porting

Currently, *packetman* is only available for SunOS 4.1.x and Ultrix 4.2a and above. Porting to other operating systems is an important extension. Systems to port to include Solaris 2.3 and IRIX.

7.7 Protocols

Though many highly used protcols are decoded, there are always more that can be added. The different versions of the RPC programs decoded (eg. NFS version 3 (Sun, 1993)) should be added. Other well used protocols should also be added: SNMP, Finger, DNS etc.

To aid in the addition of new protocols, it would be desirable to have protocol decoders created by some form of compiler. This compiler could be passed a definition of the protocol and its data structure, and produce an object file which can be linked with the body of the program.

ASN.1 could be used to give the definitions of the protocol, as it is more generally used than XDR. Compilers for ASN.1 specifications exist, one of which is Snacc (Sample, 1993). Snacc converts ASN.1 data specifications into C/C++ data structures — it performs for ASN.1 what rpcgen does for XDR. As the source for snacc is available, it could be modified to give the functionality required. This would certainly not be easy.

8.0 Conclusion

This project has extended *packetman* such that it is much more useful than the previous version. The addition of RPC and NFS — well used protocols in many LANs — has increased its usefulness dramatically. *Packetman* is the only free NFS/RPC decoder available — only commercial products can be compared to its decoding capabilities.

Packetman has proved to be useful in many ways. On the School of Computing network, AUTH_UNIX authentication is used for RPC calls without taking advantage of the AUTH_SHORT authentication type. Typical authentication parameters are 60 - 120 bytes in length, so when it is considered that RPC represents well over 60% of the bandwidth usage, using AUTH_SHORT would clearly be advantageous.

In addition, students studying the basics of RPC have been impressed with the display of “real live” RPC packets, and have found *packetman* useful towards their understanding of the protocol.

When the additions outline in section 7.0 are introduced, *packetman* will be one of the most complete protocol analysers available. The development of *packetman* will continue in the future.

9.0 References

[1] Appleton, B.(1989) *Avl-treelib-1.0* source code.

[2] Benko, G. (1992) *Packetman* v1.0 source code. Department of Computer Science, Curtin University of Technology: Perth (Western Australia).

[3] Black, U. D. (1992) TCP/IP and Related Protocols. McGraw-Hill: New York.

[4] Comer, D. E. (1991) Internetworking with TCP/IP. Volume 1; Principles, Protocols and Architectures. Prentice-Hall: London.

[5] Corbin, J. R. (1991) The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls. Springer-Verlag: New York.

[6] Curry, D. A. and J. C. Mogul (1993) The *nfswatch(8l)* manual page. Purdue University: West Lafayette.

[7] Digital Equipment Corporation. The *packetfilter(4)* manual page. Digital Equipment Corporation.

[8] Frame Technology Corporation. (1992) The *FrameMaker* Desktop Publisher. Frame Technology Corporation. San Jose: California.

[9] Hunt, M. (1992a) The *tcpview(1)* manual page. University of Washington: Seattle.

[10] Hunt, M, (1992b) *Tcpview* source code. University of Washington: Seattle.

[11] Jacobson, V. (1989) The *traceroute(8)* manual page.

[12] Jacobson, V., Leres, C., and S. McCanne (1992) The *tcpdump(1)* manual page. Lawrence Berkeley Laboratory: Berkeley (California).

[13] Kiethley, K. S. (1993) The 3D Athena Widgets (*Xaw3d*). Jet Propulsion Laboratories, NASA.

[14] McCanne, S (1991) The BPF manual page. Lawrence Berkeley Laboratory: Berkeley (California).

[15] McCanne, S. and V. Jacobson (1993) The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: Proceedings, 1993 Winter USENIX Conference, San Diego, California.

[16] Mogul, J. and J. Postel (1985) Internet Standard Subnetting Procedure. Request for comments 950.

[17] Network General Corporation (1992) The Network General Sniffer noctools entry. Newsgroup comp.networks.noctools.tools.

[18] Postel, J. (1981) Internet Control Message Protocol. Request for comments 792.

[19] Sample, M. (1993) Snacc 1.1: A High Performance ASN.1 to C/C++ Compiler. Department of Computer Science, University of British Columbia: Vancouver (Canada).

[20] Schulze, M., Benko, G. and C. Farrell (1993) Homebrew Network Monitoring: A Prelude to Network Management. Technical Report No. 9-1993, Department of Computer Science, Curtin University of Technology: Perth (Western Australia).

[21] Silicon Graphics Inc. The NetVisualyzer reference manual. Silicon Graphics Inc.: Mountain View (California).

[22] Sun Microsystems Inc. (1987) XDR: External Data Representation Standard. Request for comments 1014. Sun Microsystems Inc.: Mountain View (California).

[23] Sun Microsystems Inc. (1988) RPC: Remote Procedure Call Protocol Specification. Request for comments 1057. Sun Microsystems Inc.: Mountain View (California).

[24] Sun Microsystems Inc. (1989) NFS: Network File System Protocol Specification. Request for comments 1094. Sun Microsystems Inc.: Mountain View (California).

[25] Sun Microsystems Inc. (1990) *NIT(4P)*, SunOS 4.1.1 Reference Manual. Sun Microsystems Inc.: Mountain View (California).

[26] Sun Microsystems Inc. (1993) NFS: Network File System Version 3 Protocol Specification. Sun Microsystems Inc.: Mountain View (California).

[27] Tichy, W. F. (1991) RCS — A System for Version Control. Department of Computer Sciences, Purdue University: West Lafayette (Indiana).

Appendix A — Glossary

Abstract Syntax Notation 1: A standard for machine independent representation of data. ASN.1 is defined as part of the OSI model.

ARP: See Address Resolution Protocol.

ASN.1: See Abstract Syntax Notation 1.

Address Resolution Protocol: Used to map IP addresses to ethernet addresses. A host will broadcast an ARP message which indicates the IP address of the host whose ethernet address is wanted. All hosts will examine the message, and the specified host will reply directly to the originator with its ethernet address.

Callback: When an action of a widget is invoked (e.g. a button is pressed), the callbacks associated with that action are invoked. The *quit* button in *packetman* has a callback which closes the application. When the button is pressed, this callback is invoked and the application quits.

CSMA/CD: Carrier Sense Multiple Access/Collision Detect. Before transmitting a message, a host checks if a carrier signal is present on the wire. If not it will transmit its message. If a collision occurs (i.e. two or more hosts transmitted simultaneously) the hosts involved will wait a random time then retransmit.

DEC: Digital Equipment Corporation.

Ethernet: One type of LAN, consisting of devices connected to coaxial cable. Ethernet uses CSMA/CD technology and is a best effort delivery system.

External Data Representation: A standard defined by Sun more machine independent representation of arbitrary data structures.

Filtering: The ability to discard packets that are uninteresting or otherwise not wanted.

Frame: The ethernet term for a packet.

ICMP: Internet Control Message Protocol. Used for limited error control in TCP/IP.

IRIX: The implementation of the Unix operating system used on Silicon Graphics machines.

LAN: See Local Area Network.

Local Area Network: A physical network technology that operates at high speeds over short distances. e.g. Ethernet (Comer, 1991).

Localtalk: Also known as Appletalk. It is the inbuilt networking provided in all Macintosh computers.

Motif: One of the many "flavours" of X11 available. It is more precisely a widget set. Another widget set is the Athena (Kiethley, 1993) widget set.

Network File System: Developed by Sun to allow the sharing of files across the network. A system makes a file-system available, and other systems mount it so that it can be used as if it was a local file-system. Users using NFS mounted file-systems need not know that the files are somewhere other than their local machine. It is transparent to the user. NFS uses Remote Procedure Calls (RPC).

NFS: See Network File System.

Network Interface: The interface between a computer and the network. The network interface is used to retrieve packets from and send packet across the network. There are many types of network interfaces include Sun's *NIT* and Digital's *Packetfilter.*

Network Interface Tap: Sun's implementation of the network interface. It is very inefficient when compared to *Packetfilter* and *BPF.*

NIT: See Network Interface Tap.

OSI: Open Systems Interconnection model. A conceptual model that defines seven layers used for systems interconnection.

Packet: One protocol information unit. A conversation between two machines is made up of many packets.

Packetfilter: Digital's implementation of the network interface, available for use on Ultrix systems.

PC: Personal Computer. Usually refers to an IBM PC or compatible.

Promiscuous Mode: Normally, a network interface will only pass packets to the operating system if the packet is a broadcast or is addressed to that machine. When the interface is placed into promiscuous mode (this can only be accomplished by the super-user) the interface will pass all packets, regardless of their destination, to the operating system.

Protocol: A way by which machines may communicate with one another. Machines using a particular protocol must follow the rules it sets out for successful communication. There are many types of protocols. e.g NFS is a protocol for remote file sharing.

RARP: See Reverse Address Resolution Protocol.

Real-time: As it happens. If something is being monitored in real time then the statistics/graphics are updated constantly to display the situation of the network at that instant in time. An example of non-real-time monitoring would be a packet analyser, which looks at information that may be hours or days old. i.e. That in no way represents the network at that particular instant.

Remote Procedure Call: A mechanism developed by Sun to allow a process on one machine to call the procedures of a process on another. Conceptually, it is the same as a process calling one of its own functions. It is used to provide many networking services. e.g NFS and NIS.

Reverse Address Resolution Protocol: Used by a machine to find its IP address. The machine broadcasts a RARP message, and a host specified the RARP server replies directly to the machine telling it its IP address.

RPC: See Remote Procedure Call.

Snoop: Silicon Graphic's implementation of the network interface, used in the IRIX environment.

SunOS: The implementation of the Unix operating system used on some Sun machines.

TCP: See Transmission Control Protocol.

Telnet: Protocol for remote terminal connection.

Token-ring: The token ring network hardware produced by IBM. Access to the ring can only be accomplished if a machine has in its possession a special packet called the *token*, which is passed around the ring (Comer, 1991).

Transmission Control Protocol: Provides reliable connection orientated service for IP. Includes retransmission and error recovery services.

UDP: See User Datagram Protocol.

Ultrix: The implementation of the Unix operating system used on DEC machines.

User Datagram Protocol: Allows a process on one machine to send a datagram to a process running on another. It is unreliable as it has no retransmission or recovery systems.

Widget: The term widget is used to refer to any object in X11 applications. For example, a button is a widget, as is a pane. Widgets perform actions when something happens to them. e.g. The *capture* button in *packetman* pops up a dialog box when it is clicked.

XDR: See External Data Representation.

Appendix B — Manual Page

NAME

packetman – analyse Ethernet packets.

SYNOPSIS

packetman [**–I** *interface*] [**–xrm resource**]

DESCRIPTION

Packetman is an X based utility which analyses Ethernet packets on a retrospective basis. The trace is derived from packets promiscuously received on a local network interface. **Under SunOS:** You must be root to invoke *packetman* or it must be installed setuid root. **Under Ultrix:** *packetman* can be executed by any user, however the super-user must first invoke promiscuous-mode operation using *pfconfig*(8). Note that if *packetman* is to be invoked by the super-user then promiscuous-mode is not needed.

OPTIONS

–I interface
Use the specified interface (le0, ln0,...)

–xrm resource
Use this resource.

GRAPHICAL DISPLAY

The *packetman* display is vertically divided into thirds. The top section is the main packet trace that is stored in memory. Packets will only appear in this window if they have been captured or loaded from disk. At the top of this section is an informational heading:

 # time stamp len src addr src host–>dest host dest addr protocol stack

which can be used to identify which packets are of interest.

A packet of interest may be selected by pressing the left mouse button over the area occupied by the entry, or line, in the packet trace. The selected packet will be highlighted, and a decomposition of the packet will be shown in the lower two thirds of the display. Note that at any time the *packetman* window may be resized by the method most appropriate to your window manager, and the size of all three sub-windows may be adjusted via the red grips.

A number of menu/command buttons appear at the top of the window, these are:

Quit Filter Hosts Capture Save Load Clear

Quit gracefully exits.

The *filter* menu allows you to create a real-time protocol filter. At present, only a restricted set of filters are available. Common Ethernet protocols such as IP, Banyan, Novell etc. can be selected from the main filter menu. As each protocol is selected, it is cumulatively applied to the current filter. For example, if IP was selected, only IP traffic would be monitored. If the DECnet protocol was selected sometime later, then both IP and DECnet would be monitored. This process can continue until all protocols are selected. To clear a filter, simply select the *all* option.

Note that the *filter* menu also contains a number of sub-menus. These menus can be used to select certain sub-protocols (at present only IP/TCP and IP/UDP). These filters work in the same manner as the main Ethernet protocol filters.

The packet filtering option is far from complete. At present there is no mechanism which will allow you to examine a current filter, it is up to the user to keep track of which protocols are being filtered. It is hoped that a future release will include improved filtering capabilities. Such additions may include a custom filter which may be specified via filtering “language” similar to that in *tcpdump*, and a view or description of the current filter.

The *capture* command pops up a window requesting the number of packets you wish to capture. At present a limit of 10000 packets is imposed for any one packet trace, but you may *clear* the trace and begin a new one within the same session. A number may be entered by either hitting return after the number is typed, or pressing the *start* button. If the *start* button is pressed and the dialog contains no number or an invalid entry then *packetman* will default to the maximum packet and commence the capture. If the *start* button is used to commence the capture, it’s label will change to *stop,* which can then be used to stop the capture. The *cancel* button can be used to popdown the *capture* window **before** the commencement of the trace.

The next two buttons are used to load and save packet traces, which must be in *Network General Sniffer* format.

The *save* command will popup a dialog asking for a filename in which to save the packet trace. A filename may be entered by either hitting return after typing, or clicking on the *start* button.

Load will popup a dialog asking for a filename from which to load a previously saved packet trace. Once the trace is loaded, it will appear as if it were just captured.

Hosts will popup a dialog prompting for host names for host filtering. Two host names must be entered, separated by white space. If the filter can not be set, it probably means the host names were typed incorrectly. The *Add* button will add a host pair, the *Clear* button will clear all host filters and *Cancel* will pop down the dialog. The return key defaults to *Add.*

The *clear* button can be used to clear a trace from main memory.

FILES

/etc./ethers — Name to Ethernet address mappings.

./passwd — User ID to User name mappings. Note that this file is not needed but is highly recommended. If it does not exist then *packetman* will attempt username translation using *getpwuid.* Please note that not only is this potentially inaccurate, it can also be very slow.

This file is a derivative of the passwd file and can be created with the command “ypcat passwd|cut -d: -f1,3|sed s/:/ /g|sort -n +1”. The name of this file can be configured in *Packetman.ad.*

./group — Group ID to Group name mappings. Note that this file is not needed but is highly recommended. If it does not exist then *packetman* will attempt username translation using *getpwuid.* Please note that not only is this potentially inaccurate, it can also be very slow.

This file is a derivative of the passwd file and can be created with the command “ypcat group|cut -d: -f1,3|sed s/:/ /g|sort -n +1”. The name of this file can be configured in *Packetman.ad.*

./Packetman.ad — Applications default file for *packetman*. Note that this file is not needed, but may be used to modify the *packetman* setup with an: *xrdb -merge Packetman.ad*.

SUPPORTING LIBRARIES

X11 and Xtoolkit Libraries by MIT X Consortium
Xaw3d Library by Kaleb Keithley, NASA Jet Propulsion Labs
EzMenu Library by MIT X consortium, Dirk Grunwald and George Ferguson

SEE ALSO

interman(1), etherman(1), ethers(3), X(1), tcpdump(1), nfswatch(8)

BUGS

Certain packet filters/network taps cannot “see” traffic from the interface it is associated with. This isn’t a bug, more of a shortcoming which has no fix for *SunOS*.

NIT is not exactly efficient. Unfortunately, packetman will drop packets in SunOS.

I’m sure more bugs exist - please send bug reports to **netman@cs.curtin.edu.au**

Appendix C — Noctools Entry

NAME

Packetman

KEYWORDS

Analyzer; Ethernet, IP, NFS; Eavesdrop, X; DEC, Sun, UNIX; Free.

ABSTRACT

Packetman is an X-11 based protocol analyser that can be used to retrospectively analyse packets on an ethernet LAN. Packetman can be used for diagnosis, monitoring and trouble-shooting of network protocols under a Unix environment.

The Packetman display is divided into 3 sections ala TcpView and the Sniffer: The top section is a sequential trace of captured data; the second provides detailed analysis of a packet; and the bottom shows a hex dump of the packet.

Several well used protocols can be decoded including:

- NFS (Network File System) v2
- NIS (Network Information Service) v2
- Mount Protocol v1
- Yppasswd v1
- Sun Portmapper v2
- ICMP
- Telnet
- Arp

Packetman fully decodes Ethernet/TCP/UDP/IP headers and resolves ethernet/IP addresses to host names. RPC headers, including the authentication, are also fully decoded with the RPC replies being matched to the calls by their transaction IDs. This enables RPC reply packets of recognised protocols to be fully decoded.

Sniffer format files are supported.

MECHANISM

Packetman uses the Ultrix packetfilter on DEC systems and NIT on Sun systems, which are placed into promiscuous mode to read all packets on the network.

Several support files are used for name translations. These include /etc/ethers and derivatives of the group and passwd files which are used for ethernet/machine-name resolution and also uid/user-name and gid/group-name resolving. The information from these files is read into hashtables at start-up time.

CAVEATS

If a Dec system is operating in promiscuous mode then it is possible for a user to use Packetman to compromise network security. For example, a patient user may capture sensitive information (including passwords) and use this to compromise the network.

By default promiscuous mode is disabled.

BUGS

The NIT packet filter used on Sun systems is not efficient enough to keep up with a moderate to heavily loaded network. This became evident when the snap length was set to zero(infinity). The code has been streamlined as much as possible, but the solution seems to be BPF. Unfortunately BPF is not available to us as we do not have access to the required SunOS kernel source.

This results in some dropped packets on Sun systems.

LIMITATIONS

Under SunOS, only the Super-user can execute Packetman. On Ultrix platforms, Packetman may be executed by any user, provided the Super-user has enabled promiscuous mode using pfconfig(8c).

HARDWARE REQUIRED

SUN Sparc (SunOS 4.1.x) or DEC-mips (Ultrix 4.2a and above) workstations supporting the X11 Windowing System.

SOFTWARE REQUIRED

X11 release 4 or later.

AVAILABILITY AND CONTACT POINT FOR INFORMATION ABOUT THIS TOOL

Available via anonymous ftp from `ftp.cs.curtin.edu.au`, in the directory `pub/netman`. Binaries for SUN Sparc (SunOS 4.1.x) and DEC-mips(Ultrix 4.2a and above) are available in the subdirectories `sun4c` and `dec-mips` respectively.

Greg Barron,
Netman Development Group,

Department of Computer Science,
Curtin University of Technology, Perth, Western Australia.
Ph: +61 9 351 7666
Fax: +61 9 351 2819
email: netman@cs.curtin.edu.au

CONTACT POINT FOR CHANGES TO THIS CATALOG ENTRY

As above.

DATE OF MOST RECENT UPDATE TO THIS CATALOG ENTRY

931027