Computer Topics 591

Enhancing the user interface of *packetman*'s filtering mechanism

Mazahir Songerwala
910498E
e-mail: songerwa@cs.curtin.edu.au

September 22, 1994

Abstract

This report describes the work undertaken for Computer Topics 591: the addition of a graphical user interface to the filter selection mechanism of *packetman*. The filtering system in version 1.1 of *packetman* lacked the features required of an intuitive filtering system. This work addresses these deficiencies, and the new version of filtering allows a user to select and remove protocol and host filters with ease.

A proposal for a complete filtering system, using concepts described in McCanne and Jacobson (1992) is also provided, with a detailed description on how these concepts can be mapped onto *packetman*.

Contents

List of Figures

1 Introduction

Packet analysis is an important area of network management which allows systems administrators to study the behaviour of a particular network. Much has been written about methods of packet analysis, and many automated packet analysis tools do exist which can be used by network managers to trouble-shoot network operations. Tools which are used by many network sites today include *Sniffer*, *HP Packet Analyser*, *NetVisualizer*, *tcpdump*, and *tcpview*.

The *packetman* tool was developed by the Netman Development Group at Curtin University of Technology's Computer Science Department in 1992 (Benko, 1992), and has since undergone a major revision (Barron, 1993). A major area of *packetman* is the filtering mechanism used to capture certain types of packets. The current filtering mechanism is borrowed from *etherman* (Schulze *et al.*, 1993) and is not entirely appropriate for the filtering required for a packet analyser.

This report describes the work undertaken for the Computer Topics 591 unit, which adds an intuitive graphical user interface to the filter selection mechanism. The current state of filtering in *packetman* will be discussed, with a detailed analysis of the deficiencies of the current system and a formulation of goals for this work. This will be followed by a description of the new method of filtering in *packetman*, including a detailed discussion on the implementation details and method of use. A proposal of a complete filtering system is also provided, building upon the ideas suggested by McCanne and Jacobson (1992). Finally a description of the limitations of the current system is provided.

2 Filtering in *packetman* - Before

2.1 Background

One of the most important aspects of a packet analyser is the selection of filters available for packet analysis. A rich and wide range of packet filters provide network administrators with the flexibility to monitor a network at the very lowest level.

The functionality of a packet analyser is directly dependent on the complexity and power of the filtering available. Tools such as *tcpdump* (McCanne and Jacobson, 1992), have proven to be very useful, by providing an efficient mechanism to capture packets, as well as a powerful filtering mechanism.

2.2 Current Filtering Available

The current status of filter selection in *packetman*, comprises a menu system which services the following filter types:

```
All packets
IP (Internet Protocol)
```

```
          ICMP (Internet Control Message Protocol)
          UDP  (User Datagram Protocol)
              NFS (Network File System)
              DNS (Domain Name Server)
              SunRPC (Sun Remote Procedure Call)
              tftp
              rwho
              syslog
              talk
              ntalk
      TCP  (Transmission Control Protocol)
              ftp
              telnet
              SMTP (Simple Mail Transfer Protocol)
              Sunrpc
              finger
              NNTP (Network News Protocol)
              X400 *
              lpd *
      DEC
              DECNetIV
              LAT
              DECNetV *
              LAVC *
              MOP *
      AppleTalk *
      Novell *
      Custom Filter *

* - Not implemented
```

Currently, filter selection is a two-pronged process. Frame and protocol filters are defined by the Filter button, which displays a menu of entries corresponding to various filters. Host filtering is also offered by *packetman*, and prompts the user for input via a dialog box defined by the Hosts button. A host pair must be entered for source and destination hosts respectively. Figure 1 displays the host filtering input acquisition process.

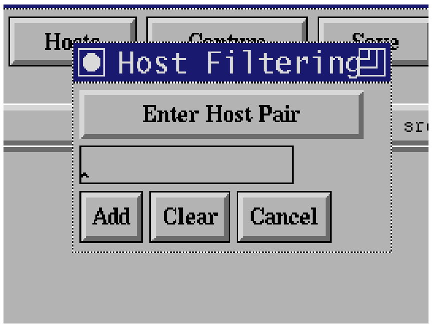

Figure 1: Host filtering in *packetman* v1.1

2.3 Deficiencies of *packetman*'s filtering mechanism

The packet and host filtering mechanisms currently existing in *packetman* are deficient, in terms of user interface design. A good user interface should display all relevant information at all times, or at least have an simple means of accessing the required information easily (Heckel, 1984).

2.3.1 Deficiencies of the protocol filtering mechanism

Currently, users must select filters from a menu, by opening the menu with a mouse click, and releasing the mouse button to signify a filter selection. The problem with this approach lies in the feedback to the user. This mechanism does not respond to the user's actions, in that no acknowledgment is given when user selects a particular filter. This feedback is important, so that users can keep track of what filters are being set.

Another deficiency of the current filter selection model is the lack of a mechanism to keep track of the filters set. The application does not maintain an internal table of set filters which the user can refer to when required. Therefore it may become difficult for a user to keep track of a large number of set filters. Clearly some scheme of maintaining a list of set filters is required.

2.3.2 Deficiencies of the host filtering mechanism

Host filters are set via a dialog box defined by the Hosts button. This dialog box prompts a user for a host pair in the form of:

```
<source>                    <destination>
```

Unfortunately, the order of hosts is implied with no form of labeling indicating the position of the arguments. Any number of host filter pairs can be set, and the dialog determines whether a host pair is indeed valid before setting the filter.

Another problem with the current host pair input scheme lies in wildcarding addresses. Currently, there is no way of specifying any source or any destination address. thereby allowing a host filter of traffic from or to a host. In other words there is no way to set a filter which requires traffic from *cain* to all other hosts. This facility is important, because it allows a system administrator to isolate network traffic for a particular host.

A problem for both protocol and host filtering, involves the resetting of all filters. Currently the technique for resetting packet and host filters lacks the ease of use and level of feedback which is required of such a process. Protocol filters are reset by selecting All packets from the filter menu (this is a problem, because the user cannot determine whether the operation was indeed successful).

The filtering mechanism used by *packetman* involves setting up a filter structure which includes the Ethernet frame type, as well as any subprotocols which may be defined. The structure involves defining a new filter frame for each type of filter (Ethernet frame) defined. Any subprotocols which are defined are set as a sublist of an filter frame . Since most of the frame types defined in the filter list are of type IP (Ethernet IP), most of the filters set will actually be defined in the protocol sublist of an IP filter frame.

This analysis of the filter structure is important because it shows a deficiency in the filtering mechanism as far as host filtering is concerned. To implement host filters for *particular* protocol filters is difficult, and requires a restructuring of the filtering data structures and mechanism.

2.4 Aims and objectives

After an analysis of the current filtering input acquisition process of *packetman*, the following changes were proposed:

1. The development a user friendly interface to filtering, which displays all relevant information when required.
2. A method of displaying all current filters set.
3. A method of clearing all filters set, including the provision of feedback to the user when this operation is carried out.
4. The extension to the host filtering scheme to allow wildcards (*) to be set for source and destination addresses.

3 Filtering in *packetman* - After

3.1 Implementation Details

The new filter selection mechanism of *packetman*, makes use of the popup mechanism provided by the X11-Windowing system. The advantage of the X system, lies in the portability of applications to other machines, and hence operating systems. In the case of *packetman*, this feature is very important, since it allows the use of the packet analyser on a variety of platforms. Currently, *packetman* has been ported to DEC-ULTRIX 4.2a and SunOS 4.1.x systems. This section will outline the interface details of the new filter selection mechanism.

3.1.1 Filter selection Popup

The filter selection popup was designed to encompass all the required facilities of a filter system. This includes:

- an easy way to define filters;
- an easy way to delete filters; and
- some manner of displaying all selected filters.

The approach chosen to address the above requirements involved the design of a comprehensive filter popup, which also provided a facility to access host filtering. A novel feature of *packetman*, is the specification of the widget hierarchies required by the application. These hierarchies can be specified in the application defaults file, and therefore may be changed without recompiling.

The widget hierarchy for the filter selection mechanism is:

```
.                                       TransientShell  filterShell
filterShell                             Box             masterform
filterShell.masterform                  Paned           masterpane
filterShell.masterform.masterpane        Viewport        pane1
filterShell.masterform.masterpane.pane1     List            pane1list
filterShell.masterform.masterpane        Viewport        pane2
filterShell.masterform.masterpane.pane2     List            pane2list
filterShell.masterform.masterpane        Viewport        pane3
filterShell.masterform.masterpane.pane3     List            pane3list
filterShell.masterform                  Paned           statusbox
filterShell.masterform.statusbox         Label           filterheader
filterShell.masterform.statusbox           Viewport        filterview
filterShell.masterform.statusbox.filterview  List            filterlist
filterShell.masterform                   Box             commands
filterShell.masterform.commands           Command         hostfilter
filterShell.masterform.commands           Command         add
filterShell.masterform.commands           Command         delete
filterShell.masterform.commands           Command         done
filterShell.masterform.commands           Command         clearall
```

The "." represents the top of the hierarchy, which allows it to be defined as a popup. Figure 2 represents this hierarchy in a graphical form.

The `GetFilterPopup` routine defined in packetman.c defines the operations for the filter selection popup, including callback bindings and initial values for the widgets used. The algorithm for this routine is:

```
void GetFilterPopup(widget,dialogname)
Widget widget;
char *dialogname;
{
   Initialise the hierarchy from the application defaults string
   Create all required widgets

   Set up the list widgets

   Add callbacks to the widgets:
          Done button             - Close popup
          Add button              - Add a filter
          Delete button           - Delete a filter
          Clear All button        - Clear host and protocol filters
          Filters selected list   - Enable deletion of filter
          First pane list         - Extract selected item
          Second pane list        - Extract selected item
          Third pane list         - Extract selected item
          Host filter button      - Invoke host filter popup

   Determine co-ordinates for filter popup

   Popup hierarchy
}
```

The filter selection popup is invoked by clicking on the Filters button of the main *packetman* window. This Filters widget defines a callback to the above routine. The following subsection describes the various callbacks used by the filtering hierarchy.

Callbacks

The Done button defines a callback which parses the list of selected filters in order to extract the information appropriate for the addition of a filter frame (located in `SetFilters` in packetman.c) The algorithm for this callback is:

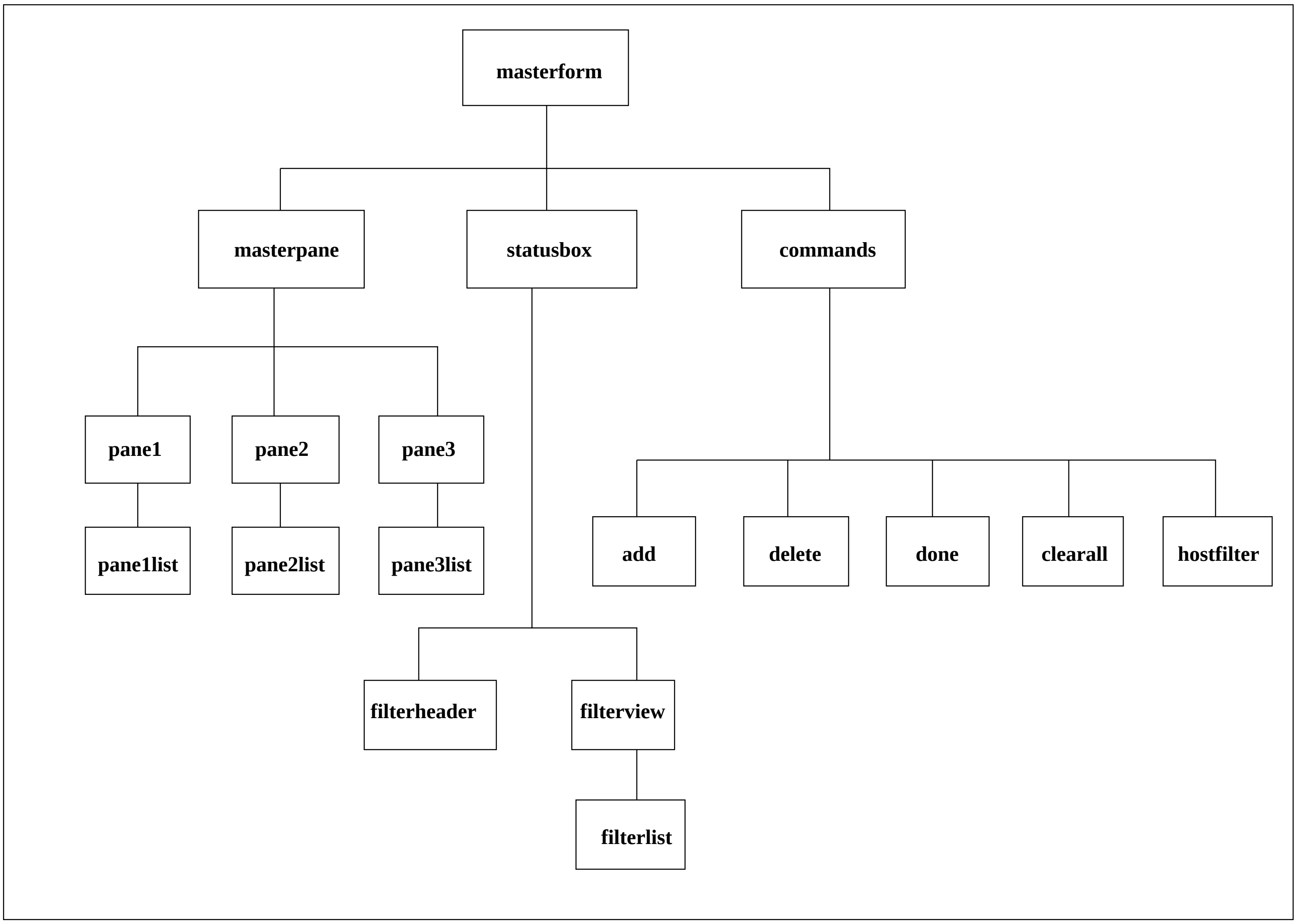

Figure 2: Filter selection popup hierarchy

```
void DoneFilter(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
   If there are filters selected

        For each filter defined
              Extract the required filter (ReadFiltersFromListArray in widgets.c)
              Set the filter (SetFilters in packetman.c)

  Destroy the Popup
}
```

The Add button callback adds a selected filter to the filter list, which is displayed in the middle of the popup. This callback is responsible for capturing the user's protocol filter selection , and adding the current source and destination hosts to form a filter definition. The algorithm for this callback is:

```
void  Add(w, client_data, call_data)
```

```
Widget w;
XtPointer client_data, call_data;
{
   If no filters have been defined, then exit

   If there is an item selected in the first pane
        Copy the item to the filter string

   If there is an item selected in the second pane
        Copy the item to the filter string

   If there is an item selected in the third pane
        Copy the item to the filter string

   Add the current host filter pair

   Copy the filter string to the filters selected list

   Update the filters selected list widget to display the new addition

   Re-initialise the list panes
}
```

The Delete button callback is responsible for removing a filter from the set of currently selected filters. The particular filter to be deleted must be selected before the Delete button becomes active. The algorithm for the Delete callback is :

```
void  Delete(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
   Obtain the filter string to be removed

   In relation to its position in the filter list, remove the entry and shift the
   remaining entries

   Re-initialise the rest of the list

   Update the filters selected list widget
}
```

The Clear All button defines a callback which removes all protocol and host filters defined by the filter selection popup. The algorithm for the callback is:

```
void  ClearAll(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
   Remove all protocol filters (remove linked list of filters)

   Remove all host filters     (remove linked list of host pairs)

   Re-initialise the filters selected list and current host pairs

   Update the filter popup to reflect these changes
}
```

The selected filters list defines a callback which makes the Delete button active if an entry is chosen. This routine must be called before any filters can be removed. The filter panes all define one callback which determines the bindings between entries in the various lists. This routine (`PrintItem`) determines which pane invoked the callback, and displays any sublists if appropriate. This routine requires modification if any new filters or filter lists are added. The algorithm for this callback is:

```
void  PrintItem(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{
  Obtain the selected item
  Obtain the ID of the calling list widget

  CASE list ID OF

      PANE 1:
               If the item selected was IP
                     Store the item selected
                     Display the IP sub protocol list in the second pane

               Else If the item selected was DEC
                     Store the item selected
                     Display the DEC sub protocol list in the second pane

               Else
                     Store the item selected
                     Re-initialise the panes

      PANE 2:
               If the item selected was UDP
                     Store the item selected
                     Display the UDP sub protocol list in the second pane

               Else If the item selected was TCP
                     Store the item selected
                     Display the TCP sub protocol list in the second pane

               Else
                     Store the item selected
                     Re-initialise the panes
      PANE 3:
               Store the item selected

   ENDCASE
  }
```

The Hosts button defines a callback which invokes the host filter selection popup. This routine (`GetHostPair`) is defined in widgets.c, and sets up its own widget hierarchy. The widget hierarchy for the host filter selection is:

```
.                               TransientShell  hostShell
hostShell                       Box             hostform
hostShell.hostform               Label           hostlabel
hostShell.hostform               Box             dataentry
hostShell.hostform.dataentry      Label             source_l
hostShell.hostform.dataentry      AsciiText         source
hostShell.hostform.dataentry      Label             dest_l
hostShell.hostform.dataentry      AsciiText         dest
hostShell.hostform              Box              commandBox
hostShell.hostform.commandBox    Command           addbutton
hostShell.hostform.commandBox    Command           clearbutton
hostShell.hostform.commandBox    Command           donebutton
```

The "." represents the top of the hierarchy, which allows it to be defined as a popup. Figure 3 represents this hierarchy in a graphical form.

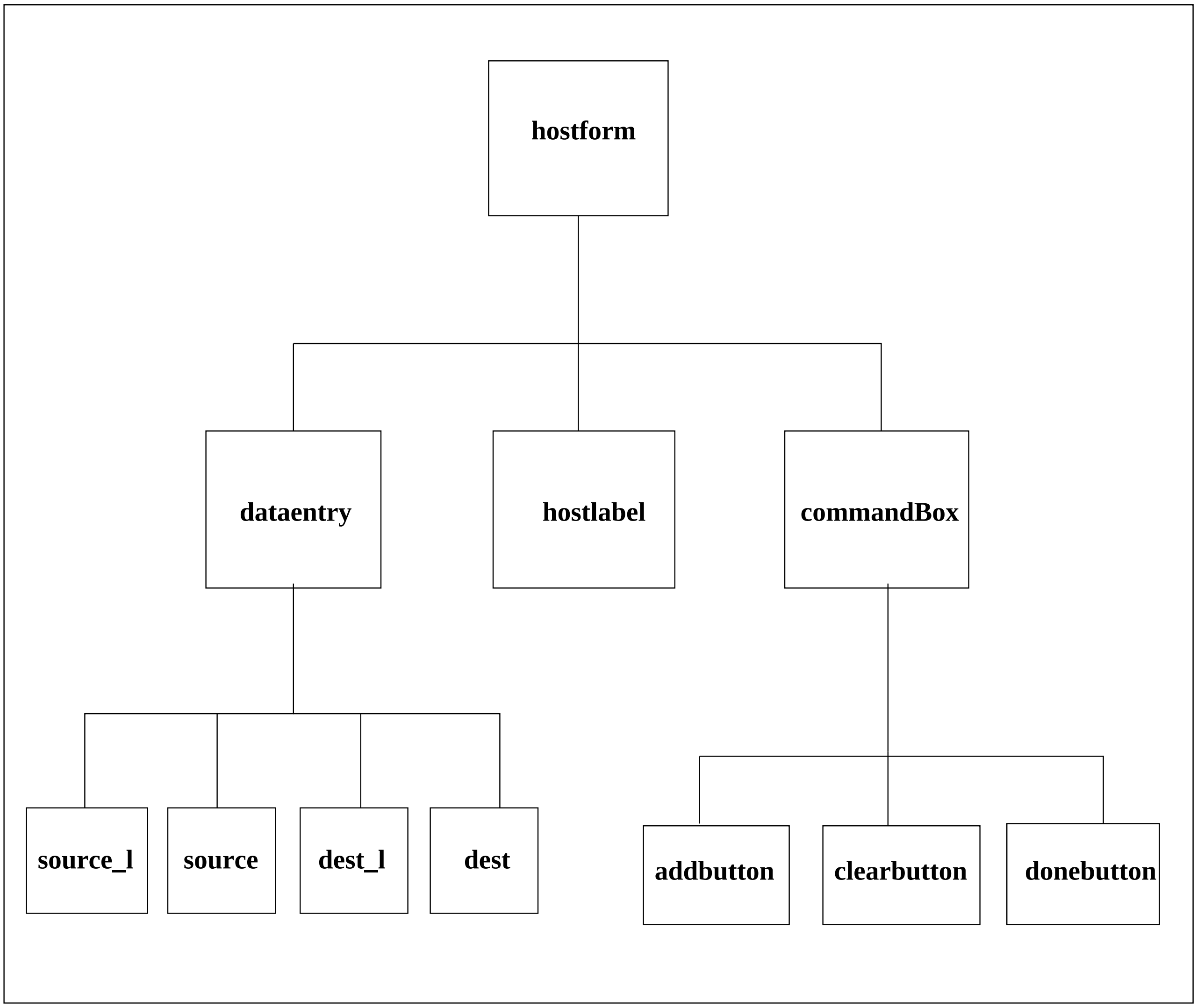

Figure 3: Host selection popup hierarchy

The *GetHostPair* routine defined in widgets.c defines the operations for the host filter selection popup, including callback bindings and initial values for the widgets used. The algorithm for this routine is:

```
void GetHostPair(widget,dialogname,func1, func2)
Widget widget;
char *dialogname;
void (*func1)(), (*func2)();
{
   Initialise the hierarchy from the application defaults string
   Create all required widgets

   Set up the ascii text widgets

   Add callbacks to the widgets:
          Done button            - Close popup
          Clear button           - Clear all host filters
          Apply button           - Parse hosts

   Determine co-ordinates for filter popup
```

```
    Popup hierarchy
  }
```

The Done button callback is responsible for setting the host filter in the filters selected list, and removing the popup hierarchy. The latest pair of input host names is associated with the "All packets" filter, and this is added to the list of selected filters. This is done so that users can keep track of host filters set when no protocol filters have been defined. That is, if the user wishes to select only host pair filters, the list will still maintain these entries in order to meet the aim of displaying all relevant information at all times. The algorithm for this routine is:

```
void AddHostPair(widget,client_data,call_data);
Widget widget;
XtPointer client_data, call_data;
{

  Copy the host pair into a temporary buffer, which contains
   the ``All packets'' filter

  Copy the temporary buffer to the filters selected list

  Update list parameters

  Update the list widget to display the new entry

  Destroy the popup hierarchy
}
```

The Clear button removes all the set host filters (i.e. destroys the linked list of host filters). The Apply button defines a callback which actually sets up a host filter in the linked list of host filters.

A new feature of host filtering in *packetman* is the use of the * wildcard. This wildcard denotes any host, enabling definitions such as source - *, * - destination, and * - *. This routine (`ParseHostsNew`) is defined in packetman.c and has the following algorithm:

```
void ParseHostsNew(widget, client_data, call_data)
Widget widget;
XtPointer client_data, call_data;
{
  Allocate space for the source and destination variables

  Obtain the values for the source and destination hosts from the Ascii text
   widgets

  Copy the source and destination hosts to the host pair global variables

  If the host pair is invalid (checked by a call to set_host_filters in filter.c)
          Display error message, and re-prompt user
  Else
          Re-initialise the source and destination hosts, and wait for more input

}
```

The `set_host_filters` routine in filter.c determines whether the input host pair exists on the current network. This check is carried out with a call to the `ether_hostton` function, which returns the Ethernet address of a given hostname. If the host name does not exist the call fails. If the hostnames exist, then a host filter is added to the linked list of host filters. The algorithm for this routine is:

```
int set_host_filters(src, dst)
char *src, *dst;
{
   Allocate memory for host filter node

   If the host pair was * -> destination
     Determine whether the destination exists

   Else If the host pair was source -> *
     Determine whether the source exists

   Else
     Determine whether the source and destination exist

   If the hosts are invalid
     Destroy the filter node
     Exit

   Set up node in the linked list
}
```

Other routines which have been added/modified include:

`packetman.c :  initLists(), SetFilters(), HostNew(), Filtering();`

`widgets.c :  RemoveTopShell(), BlankList(), ReadFiltersFromListArray(), InitListFromString();`

`filter.c :  nm_filter_match_packet();`

The `initLists` routine reads in the various lists defined in the application defaults file. Other initialisations of global variables also takes place in this routine. The `SetFilters` routine performs the steps required to add a chosen filter to the filter list, which is used in the filter-packet matching process, during packet capture. This involves setting up the filter frame with the appropriate arguments, and calling the `nm_filter_add_filter_frame` routine defined in filter.c The `HostNew` routine is used to setup the callback for the Hosts button on filter window. The `Filtering` routine similarly defines the callback for the Filter button on the main *packetman* display. When called, Filtering invokes the routine which displays the filter popup (`GetFilterPopup` in packetman.c).

The `RemoveTopShell` routine was taken from *etherman* (Schulze *et al.*, 1993), and contains code which removes a popup hierarchy. The `BlankList` routine fills a list with a number of blanks, defined by the BLANKS constant. The `ReadFiltersFromListArray` routine parses a chosen filter entry and extracts the actual filter required. For example, if the filter chosen was:

```
        *          *              IP/UDP/NFS
```

the routine would return the NFS portion of the filter. The `InitListFromString` routine parses the string obtained from the applications defaults file, which defines a particular list. Spaces are used as separators, and therefore entries spanning multiple words must be separated by the underscore character (_).

The `nm_filter_match_packet` routine in filter.c has been modified to allow wildcard host filtering. Before protocol filtering occurs, the host filter list is checked (according to the format of the host pair (i.e. source - *, etc.)). As soon as a packet matches a host pair, protocol filtering begins.

3.2 The new filter selection mechanism

Previously, the filtering mechanism provided by packetman provided a large range of filter selections, but lacked the user feedback required for such an operation. Barron (1993) notes that the filter selection process must be modified to encompass the power of the available filters, and provide adequate feedback to the user during the selection process. Barron (1993) proposed a model which would handle such requirements, and this model forms much of the basis for this work. The area of host filtering was also lacking in the previous version of *packetman*. The mechanism allowed the addition of specific host pairs, but did not provide a means of displaying the set hosts to the user.

The new filter selection mechanism in *packetman* builds upon the model proposed by Barron (1993), by providing an intuitive format for obtaining filter selection input, host filter input, and displaying a list of set filters, which is updated with user actions. This model allows the user to select particular filters easily, and determine which filters are set at that particular time.

The filtering mechanism currently in place, was designed with certain key objectives in mind:

- To develop an interface which facilitates the protocol filter selection process.
- To provide adequate feedback to the user, in the form of a list which stores the selected filters.
- To provide the ability to invoke the host filter selection mechanism as a logical extension of the filtering popup.
- To provide the ability to remove all selected host and protocol filters.
- To provide the ability to delete a selected filter.

The new filtering selection mechanism of *packetman*, meets all of the objectives listed above, whilst maintaining simplicity. Figure 4 illustrates the new method of obtaining user input for filter selection.

The filter popup box is segmented into three main sections:

1. Protocol list panes;
2. Filters selected list; and

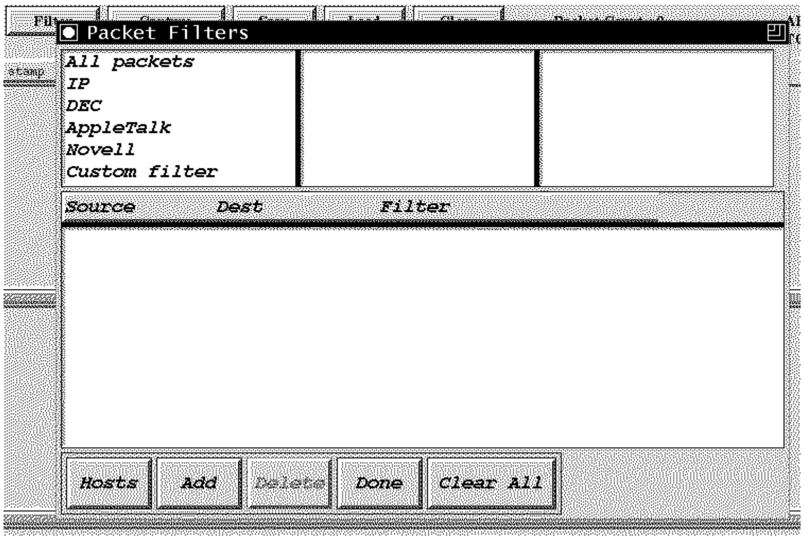

Figure 4: New filter selection popup in *packetman*

3. Command buttons.

The protocol list panes define the **protocol** hierarchy for particular filters. For example, an *ftp* filter must be defined in terms of its protocol hierarchy: *IP/TCP/ftp*. This selection method involves logically following the hierarchy from the root. By selecting the parent protocol (*IP*), followed by the middle protocol (*TCP*), and finally the required protocol (*ftp*), the filter can be selected. Each filter pane is linked to the others, with respect to the entries defined in the filter lists.

The filters selected list, displays all the currently set filters. The format for these filters are:

```
Source          Destination          Filter
```

The filter selected list can have items deleted from it at any time, and may also be cleared completely.

The command buttons defined carry out the following functions:

Hosts : Clicking on this button invokes host filter selection popup, which prompts the user for source and destination hostnames.

Add: Clicking on this button adds a filter to the selected filters list, with respect to the last set host pair and the currently selected protocol filter.

Delete: Clicking on this button deletes a particular entry from the selected filters list. Note that this button can only be used, when an item in the selected filters list is highlighted (i.e selected).

Clear All: Clicking on this button clears all host and protocol filters which have been defined.

Consecutive invocations of the filter selection popup will maintain the filters selected list, so that users can keep track of the current set of filters in use.

3.2.1 Instructions for selecting filters

To best illustrate the filter selection mechanism, consider the example of setting the *NFS* (Network File System) filter. *NFS* is a member of the *UDP* protocol, which is itself a part of *IP*. Therefore the protocol hierarchy of *NFS* is *IP/UDP/NFS*. The selection and setting of this filter involves the following steps:

1. Select the root protocol family (IP) by clicking on the appropriate entry in the first filter pane. Refer to Figure 5.

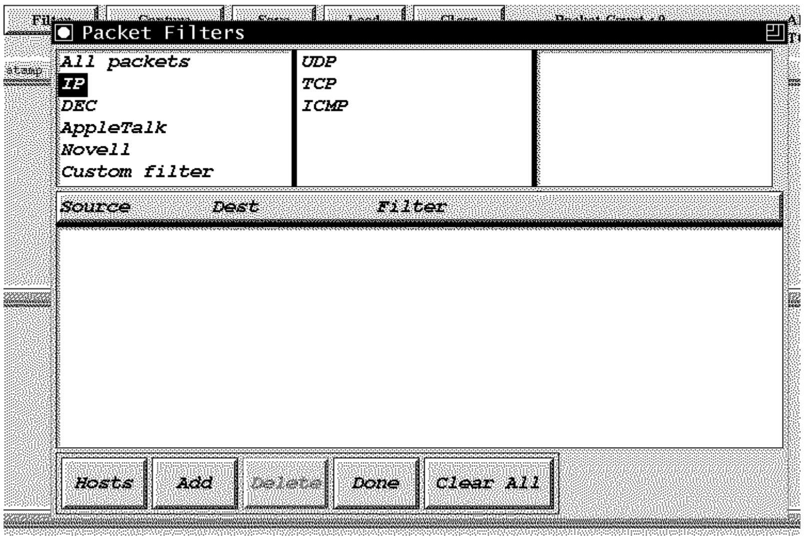

Figure 5: Selection of IP

2. Select the next level in the protocol hierarchy (UDP) by clicking on the appropriate entry in the second filter pane. Refer to Figure 6.

3. Select the NFS entry in pane 3, and add the NFS filter by clicking on the Add button.Refer to Figure 7.

Currently, upto 20 filters can be set, although this limit is configurable (via the LISTLIMIT constant defined in packetman.c). Note that the source and destination entries are *'s or wildcards, because no specific host filters have been set.

Host filters are set by invoking the host selection popup defined by the Hosts button. Figure 8 illustrates the format of the host popup.

The host popup prompts for a source and destination address, and is initialised to any hosts (using the * wildcard). Users may select any combination of hosts and wildcards (eg. source - *, * - destination, * - *, source - destination).

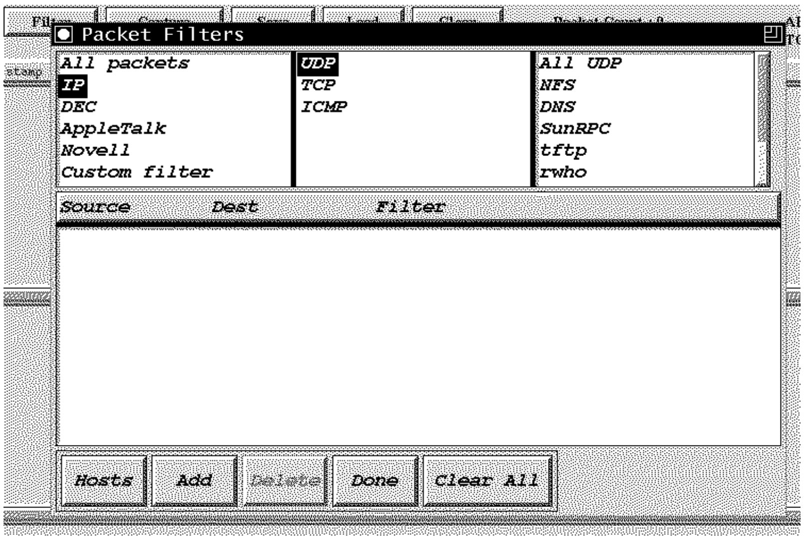

Figure 6: Selection of UDP

Host filters are added by clicking the Apply button. Once this occurs, the application verifies that the selected hosts do in fact exist on the network. If there is an error, the popup displays an error message and does not set the host filter. Figure 9 illustrates this message.

If the host selection was successful, the source and destination hosts are reset to the * wildcard, and the popup waits for more input. The user may add more host pairs, but this is not recommended, because the filters selected list only reflects the last selected host pair for a particular filter. Therefore, it is advised, that only one filter be set at a time, per protocol filter. This allows the filters selected list to maintain its consistency. The Clear button clears all host filter pairs which have been defined. The Done button exits the host popup (but does not set any filters. This feature is defined by the Apply button).

Upon exit, the filters selected list is updated to display the latest host filter added. This allows the application to maintain a list of all filters at all times. Note that the addition of more than one pair during the host pair input, will only result in a filter being displayed for the last pair (in the filters selected list). The default protocol filter associated with a host pair is the "All packets" filter. Any subsequent protocol filters set will also be associated with the latest host pair set. Therefore, users may delete the entry corresponding to the "All packets" host filter and maintain any other filters set with a particular pair.

3.2.2 Adding Filters

The new filter selection mechanism facilitates the addition of individual protocol filters and entire protocol lists. Either method requires the modification of certain sections of the application code.

Individual filters

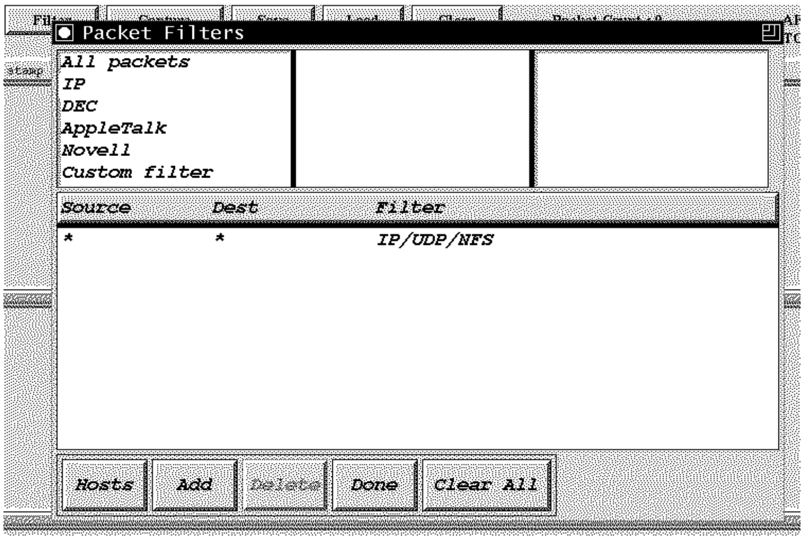

Figure 7: Addition of the IP/UDP/NFS filter

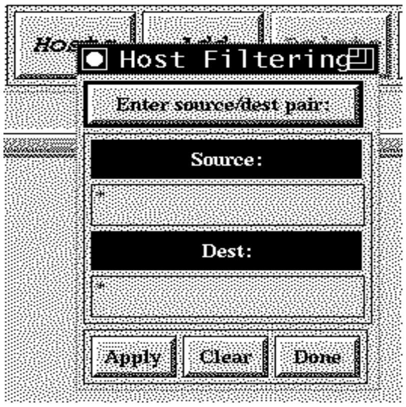

Figure 8: Format of the host filter popup

Adding an individual filter to the existing scheme (eg. adding a Custom UDP port filter) involves the modification of two aspects of *packetman*: the application defaults file and the `SetFilters` routine defined in packetman.c

1. Application Defaults File
   Adding a new filter (Custom UDP port) involves updating the list entries in the application defaults file (Packetman.ad) Currently the lists are defined as follows:

```
Packetman.udpSubProtos: \
                        All_UDP \
                        NFS \
                        DNS \
                        SunRPC \
                        tftp \
                        rwho \
                        syslog \
                        talk \
                        ntalk \
                        Custom_UDP_port
```

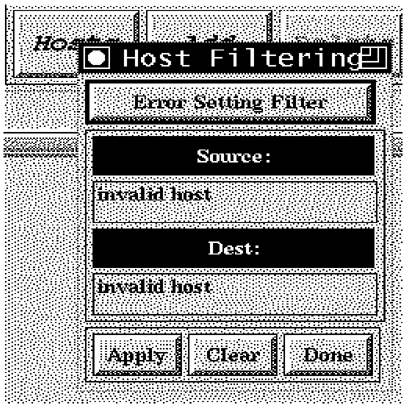

Figure 9: Error in host filter selection

The underscore character (_) is required for any filter entries which span multiple words. This is required for the list parsing routine, which has no other way of determining whether words form one or multiple filter entries.

2. `SetFilters` (packetman.c)
   Adding a new filter also requires the modification of the `SetFilters` routine found in packetman.c. An entry in this routine is in the following form:

```
if (!strcmp(test,"Custom UDP port"))
{
  frame_id = ETHERTYPE_IP;
  .....
  filter = nm_filter_add_frame_filter(...);
}
```

Filter lists

Adding a new protocol list (eg. adding a subprotocols list to the Custom Filter item of the root filter hierarchy) to the existing scheme involves the modification of three main areas of the application code: the application defaults file; and `SetFilters` and `PrintItem` in packetman.c.

1. Applications Defaults File
   Adding a new protocol list involves including the protocol list definition is the application defaults file (Packetman.ad):

```
Packetman.newSubProtos: \
                        Filter1 \
                        Filter2 \
                        Filter3
```

2. `PrintItem` (packetman.c)
   The `PrintItem` routine must be modified to reflect the bindings for the new sub protocol list. For example if **newSubProtos** was a list defined from Custom Filter, the required modifications involve updating the PANE1 section of the code, which defines the operations for the first pane:

```
case PANE1: {
                ....
                if (!strcmp(list_struct->string,"Custom Filter"))
                {
                 filterpane1 = XtNewString(list_struct->string);
                 /* Display the newSubProtos list in the second
                    pane. */

                 XawListChange(pane2list,newSubProtos,nnewSubProtos,-1,True);
                 XawListChange(pane3list,emptylist,nemptylist,-1,True);
                 break;
                }
```

3. `SetFilters` (packetman.c)
   The newSubProtos entries must be defined in the `SetFilters` routine so that entries in the list actually do add filter frames to the filter structure. This involves a *strcmp* check for each entry, and code to add filters for each entry

```
else if (!strcmp(test,"Filter1")) {
          frame_id = ETHERTYPE_CustFilt;
          filter = nm_filter_add_frame_filter(
                        filter, &frame_id, NULL, NULL, NULL, 0);
}
```

4 A Complete Filtering System - Proposal

4.1 The need for a complete filtering system

The current filtering mechanism in *packetman* relies on a rather simplistic form of obtaining user input and performing the filtering operations. The application reads "chunks" of packets into a buffer, which is then filtered via an application defined filtering routine. This process is inefficient and slow. The efficiency and speed of a packet analyser are of primary importance, because network administrators require information as it arrives to determine the cause of faults or performance degradation. Therefore, a better technique for filtering is needed, so that the inefficiencies of storing each packet in a buffer, and using an application-defined packet filter are removed.

Packet filters form the gateway to the communications medium, from which packets can be received and sent. Every packet filter is equipped with functionality to allow the creation of an access point to the network, and the reading and filtering of packets. Most packet filters have the advantage of being very quick and efficient, because they perform at the very lowest level of the system architecture.

Typically, an application formulates a request which is passed to the operating system kernel. The kernel then calls the packet filter in relation to the request made. The packet filter returns information (i.e. packets) relating to the request to the kernel, which then passes the results back to the application program. Therefore, ideally an application such as a protocol analyser should take advantage of the kernel support for packet filtering, rather than implementing its own technique.

packetman currently carries out its own filtering, where the packet filter is simply used to read packets into a buffer. An pre-defined number of packets are read into this buffer, and passed to the application filtering system. These "chunks" of packets are then matched according to the user's filter set (i.e. hosts and protocols). This is inefficient because application must carry out a large packet copy operation before processing can occur. This process is time and resource-intensive, and packets still crossing the communications medium may be dropped, thereby reducing the effectiveness of the packet analyser as a whole. By carrying out the filtering process at the kernel level, a packet analyser can only copy those packets which are necessary (i.e. those which are matched). The overhead of copying the buffer, and performing customised filtering is removed.

Currently, *packetman* can use the *packetfilter* interface for DEC-MIPS ULTRIX systems; and *NIT* for SunOS systems. However, the efficiency of various packet filters differs. For instance, McCanne and Jacobson (1992) and Jacobson (1992) note that the performance of *bpf* is 100 times better than that of NIT.

An ideal filtering system would involve the application program obtaining the users packet filtering request followed by a conversion into an intermediatary form which is machine independent. This machine independence is important because it allows the applications programmer to keep a consistent structure to filtering data. Therefore, the machine dependent routines (for the various packet filters) only require code to convert the independent data structure into a packet filter filtering request. Most packet filters define their operations in a pseudoassembly language. Therefore the machine dependent routines are responsible for converting the machine independent representation into such a form.

Existing packet analysis tools which take advantage of such a form of filtering include *tcpdump* and *tcpview*, developed at the University of California at Berkeley. These applications use the operations defined by the *bpf* packet filter to take advantage of kernel-level filtering. *tcpdump* obtains user requests and converts them into a machine independent representation, which is then converted to the appropriate *bpf* instructions. These instructions are then passed to the kernel which carries out the required operations, and passes the results back to the application. *tcpview* adds a graphical user interface to the functionality of *tcpdump*, thereby facilitating the use of the tool.

In addition, *tcpdump* provides a filtering language which defines the functionality of the tool. Such filtering languages are important because they allow both simple and complex filters to be defined in an intuitive format. User inputs can be parsed and converted to a machine representation for use by an application.

Therefore, a complete filtering system incorporates three main sections:

1. A filtering language
2. A machine independent representation
3. A machine dependent representation

The following section will discuss the proposed changes to *packetman* to add the above features into the system.

4.2 Proposed filtering system for *packetman*

4.2.1 Filtering Language

A filtering language allows users to specify filters using a certain set of fixed constructs. Since some filters can be very complicated, it is important to make sure that the input is consistent and correct. This requires the development of a parser which will examine user input and extract the required information for the filtering mechanism.

Tools do exist which allow languages to be defined according to user specified syntax and semantics. Most notable are *lex* and *yacc*. *lex* is a lexical analyser, which details the constructs and keywords of valid language statements. *yacc* is a semantic parser which associates operations with particular sets of lexical constructs.

In the case of packet filtering, a language must define host filtering, protocol filtering and logical operatives. Logical operators are used to provide users with an intuitive format for specifying filter predicates.

A prototype language for *packetman* has been developed, which parses filtering statements successfully, and returns the appropriate information required for a machine independent structure. The parser was developed using *lex* and *yacc* on a UNIX system.

The language is made up of the following keywords, which define the types of requests required by the system:

```
HOST            Defines the host keyword
PROTO           Defines the protocol keyword
SRC             Defines the originating host
DST             Defines the terminating host
AND
OR              Define the valid logical operatives
NOT
```

The following example illustrates the use of the filtering language to request all packets between hosts *cain* and *abel* (bi-directional), or any packets which are part of the Reverse ARP (RARP) protocol:

```
((host cain and host abel) or proto RARP)
```

The following example requests all packets originating from host *sodom* which are part of the IP protocol:

```
(host src sodom and proto IP)
```

The use of brackets avoids the problems of precedence, and forces the user to think in valid logical terms. This design issue leads to a better filtering language (less prone to errors), and a reduction in complexity of the parser. Implementing precedence can lead to filtering which is inconsistent with the user's perceived intention. For example, the following example shows the ambiguity of a host filtering request:

```
(host cain and host dst abel and host marsh)
```

The above statement is ambiguous because it it is difficult to determine which predicates belong together. Brackets help clarify the expression for the application and the user.

The lexical definition for the filtering language (as specified in lex) is as follows:

```
/* {\it packetman} host and protocol filtering lexical analyzer */
%{
#include <stdio.h>
#include "yydefs.h"
#include "y.tab.h"

int   yyline =  1; /* Current line number */
%}

%%

[\t ]                    ;         /* Ignore white space */
[\n]                    { yyline++; return NEWLINE; } /* Count newlines */
[Aa][Nn][Dd]            return AND;
[Oo][Rr]                return OR;
[Nn][Oo][Tt]            return NOT;
[Hh][Oo][Ss][Tt]        return HOST;
[Pp][Rr][Oo][Tt][Oo]    return PROTO;
[Tt][Cc][Pp]            return TCP;
[Uu][Dd][Pp]            return UDP;
[Ii][Pp]                return IP;
[Aa][Rr][Pp]            return ARP;
[Rr][Aa][Rr][Pp]        return RARP;
[Ii][Cc][Mm][Pp]        return ICMP;
[Dd][Ss][Tt]            return DST;
[Ss][Rr][Cc]            return SRC;
[a-zA-Z][a-zA-Z0-9]*    { yylval.str = strdup(yytext) ; return IDENTIFIER; }
"."                     return DOT;
"("                     return LBRACK;
")"                     return RBRACK;
"!"                     return NOT;
"&"                     return AND;
"|"                     return OR;
"~"                     return NOT;
"&&"                    return AND;
"||"                    return OR;
.                       yyerror("lexical error"); /* anything else */
%%
```

The above specification is broken up into a format which defines the keyword or token and the corresponding operation for the token. For example, the token host, returns a constant to the calling application. All keywords can be typed in any combination of upper- or lower-case letters. In addition, certain punctuation marks have been defined for logical operators.

The above lex code, will parse a string and extract the token found. The results of the lexical parse can be passed to a semantic parser such as *yacc* to define the rules of a language.

yacc defines the grammar for a particular language in BNF (Backus-Nuar) form. The yacc specification for the filtering language is:

```
/* {\it packetman} filtering expression parser source */
%{
#include     <stdio.h>
#include     <stdlib.h>
#include     <string.h>

char * host;      /* Name of the host */
char * IP_addr;   /* IP_address */
%}
```

```
%union
{
    char *  str;
}

%token        AND,OR, NOT
%token <str>  IDENTIFIER
%token        NEWLINE,DOT
%token        LBRACK,RBRACK
%token        HOST
%token        PROTO
%token        TCP,UDP,IP,ARP,RARP,ICMP
%token        DST, SRC

%type <str>   InetName

%%
prog : Expression NEWLINE
           { return 0; }
     ;

Expression : LBRACK InetExpression OR ProtoExpression RBRACK
           | LBRACK InetExpression AND ProtoExpression RBRACK
           | LBRACK ProtoExpression OR InetExpression RBRACK
           | LBRACK ProtoExpression AND InetExpression RBRACK
           | InetExpression
           | ProtoExpression
           ;

InetExpression: NOT InetExpression
              | LBRACK InetExpression OR InetExpression RBRACK
              | LBRACK InetExpression AND InetExpression RBRACK
              | LBRACK InetExpression RBRACK
              | InetName
              | SpecificDirec
              ;

InetName: HOST IDENTIFIER
             {  /* Code to determine whether the host exits */ }
        | InetName DOT HOST IDENTIFIER
        ;

SpecificDirec: HOST DST IDENTIFIER
                 { /* Code to determine whether the host exits */  }
             | HOST SRC IDENTIFIER
                 { /* Code to determine whether the host exits */  }
             ;

ProtoExpression: NOT ProtoExpression
               | LBRACK ProtoExpression OR ProtoExpression RBRACK
               | LBRACK ProtoExpression AND ProtoExpression RBRACK
               | LBRACK ProtoExpression RBRACK
               | ProtoName
               ;

ProtoName: PROTO TCP
         | PROTO UDP
         | PROTO IP
         | PROTO ARP
         | PROTO RARP
         | PROTO ICMP
         ;
%%
```

The above yacc specification defines the format for a logical filtering expression. The LBRACK and RBRACK tokens represent brackets, which are used to make the expression logically valid. To illustrate the format of a grammar, consider the following excerpt:

```
InetExpression: NOT InetExpression
              | LBRACK InetExpression OR InetExpression RBRACK
              | LBRACK InetExpression AND InetExpression RBRACK
              | LBRACK InetExpression RBRACK
              | InetName
              ;

InetName: HOST IDENTIFIER
            {  /* Code to determine whether the host exits */ }
        | InetName DOT HOST IDENTIFIER
        ;
```

The above constructs specify the format of a host filtering predicate. Each subexpression (InetExpression) is defined in terms of itself, with varying combinations of logical operators. The expression is made up of the InetName construct which defines the actual tokens returned by the lexical parse. InetName is defined as being a HOST keyword followed by an IDENTIFIER. The identifier can be used to determine whether a host is valid (i.e. is indeed the name of a host on the network). In this way, and expression can be defined from the most abstract level (prog), to the very basic constituents (InetName, ProtoName, and SpecificDirec)

The parser currently works for logical filtering expressions, and returns an error for ill-formed or inconsistent expressions. The *yacc* specification can pass out data obtained from the parse, to the next stage of the filtering process: conversion into a machine-independent representation.

4.2.2 Machine Independant Representation

Once a valid filtering expression is obtained, a mechanism to represent the clauses in the expression is required. Machine independence is an important factor, because it facilitates the use of many different packet filters and platforms. By maintaining a consistent intermediate representation of a filter expression, machine dependent routines can be added when required. This enhances the portability of the application.

There are two approaches, or models which can be applied to the problem of representing a logical expression : the parse tree method, and the control flow graph method.

The parse tree method, has been traditionally used to represent grammars and languages. A logical expression can be split into its various operands and operators, and represented in a binary tree, which can then be traversed, during the filter matching process. For example, the expression:

`((host cain and host abel) or proto tcp)`

can be represented by the parse tree illustrated in Figure 10.

Thus when the filter-packet match occurs, the packet is parsed with respect to the above representation. This method is inefficient and may require the traversal of the whole tree structure before a packet is matched. McCanne and Jacobson (1992) propose the control flow graph (CFG) method, which has a significant performance advantage

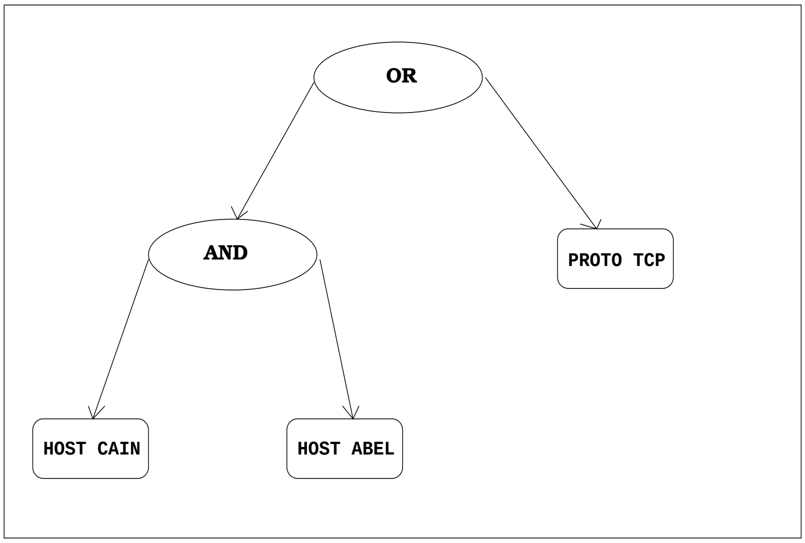

Figure 10: Parse Tree representation of `((host cain and host abel) or proto tcp)`

over the tree-based model representation. Where parse trees may require multiple parses on the same packet (eg. match host, then match proto, etc.), the control flow graph method, builds parse information into a graph structure. This method allows a particular packet to be rejected or accepted at the earliest possible point.

The CFG method, consists of a graph of nodes, which are connected by two pointers (or branches). These pointers represent true and false respectively. In relation to the proposed filtering scheme, a node would have the following structure:

```
Label
Predicate
     Host | Proto
Identifier
TRUE pointer
FALSE pointer
NEXT pointer
```

A proposed C definition for the above structure is:

```
typdef struct pred_node
{
    int         label;      /* Used in the parse */
    struct pred
    {
           char type;       /* 'h' - host, 'p' - proto */
           char *data;      /* IDENTIFIER */
    }
    struct pred_node *T;    /* True pointer */
    struct pred_node *F;    /* False pointer */
    struct pred_node *next; /* Pointer to next node */
}
```

This definition defines the structure of the list of clauses/predicates which will be formed when the parsing process takes place. Each node is comprised of:

1. a label. This is used in the pseudo assembly generation process (see next section), which requires the use of labels. Each label can be assigned sequentially.

2. a predicate check. This is used during the filter parsing process. That is, if proto tcp was entered, the predicate would be:
```
predicate.type = 'p';
predicate.data = "TCP";
```

3. a true pointer. This pointer is used as a connection to the next predicate, if the current predicate is **matched**. For example, if host *abel* was entered, then predicate.type = 'h', and predicate.data = "abel", would represent the predicate values for the node. If this predicate is true for a particular packet (i.e. the packet is either from or to abel), then the TRUE pointer is traversed to the next node of the graph.

4. a false pointer. This pointer is used as a connection to the next predicate, if the current predicate is unmatched. This is especially used in the OR or NOT cases.

5. a pointer to the next node of the list.

A proposed scheme of implementing the CFG, based on the above node structure, would be to create nodes during the yacc parse. That is, every occurrence of a host or proto identifier would result in the creation of a node with certain default characteristics. These include: true and false pointer pointing to final TRUE and final FALSE, the next pointer pointing to NULL, the previous node's next pointer pointing to the newly allocated node, and the predicate data initialised. Final TRUE and final FALSE are globally nodes, which have no further links to other nodes. They must be created before the parse takes place, and are the final states of the parse.

On occurrence of an operator such as AND, OR, and NOT, the true and false pointers are modified, such that the matching partner predicate (i.e. a AND b, where a and b are partners) is included in the CFG. For example, if the filter expression is `((host abel and host cain) or proto tcp)`, then the true pointer of predicate abel must point to predicate cain. Therefore, during the filter-packet matching process, if the packet's source or destination address is abel, then the true pointer is followed to match predicate cain. If this is also true, then the packet can be accepted (by following the true pointer to final TRUE). However, if the packet does not match, then the false pointer is followed to the proto tcp predicate. Figure 11 describes the CFG structure.

The complexity of the graph increases with the length of the expression. Consider the yacc definition in Section 4.2.1 If an expression is complex and lengthy, the yacc definition must take into account the breakdown process of the expression into the key predicates, including all relationships between predicates. By including an OR proto TCP in the above example, the nodes for predicate cain and predicate abel must modify their false pointers to reflect the new clause. This process of constructing a

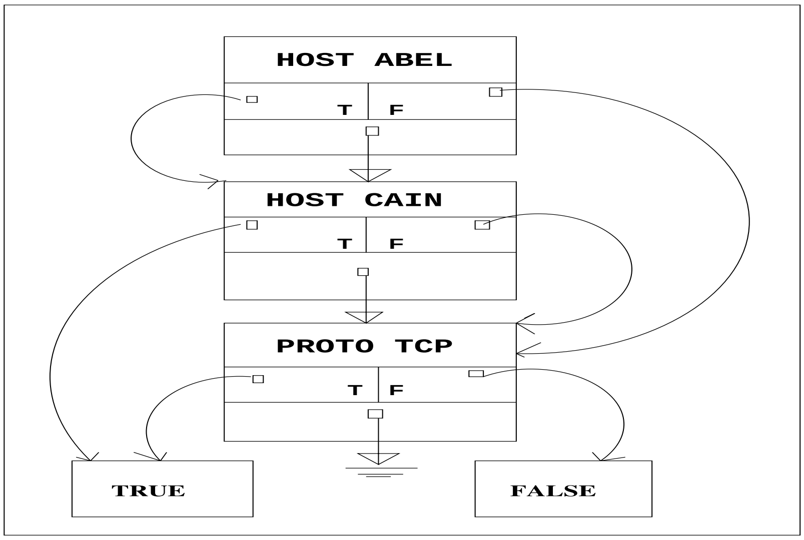

Figure 11: Control flow graph representation of `((host cain and host abel) or proto tcp)`

CFG to match complex expression is non-trivial (McCanne and Jacobson, 1992), and not in the scope of this unit.

4.2.3 Machine Dependent Representation

The final stage of the proposed scheme involves generating a pseudo-assembly language generator for various packet filters. The advantage of the machine-independent representation of a filter expression, lies in the ease in which new routines can be added for new packet filters. These new routines require mechanism which extract the relevant details from the CFG, and generate instructions appropriate for the packet filter used. McCanne and Jacobson (1992) adopted such an approach for *tcpdump*, which uses the CFG method to generate *bpf* pseudo-assembly language code. An ideal method for the *packetman* scheme, would be to extract the relevant details, and generate instructions for each node of the CFG. These instructions can then be added to a buffer space, and joined with other instructions (in relation to the true and false pointers, and the labels of each node). This buffer space can then be passed to the operating system which will execute the packet filter on the instructions. Any packets matching the instructions will be returned to the application. This represents a large increase in speed and efficiency of the packet filtering process, because the operating system is responsible for carrying out I/O and communications requests.

5 Limitations of the current system

The new interface for *packetman* was designed with a more complete filtering system in mind. The current mechanism for physically setting filters in the system does not entirely map onto the interface implemented. Section 2.3 discussed the format of the filter structures used by *packetman*, where filters were allocated according to the main frame type, and later by sub protocols and ports. The user interface for filter selection is design to display all active filters at any one time. This is represented in the filters selected list in the middle of the filter popup. This list has a format which includes the source address, destination address and the filter chosen. This representation supports the concept of setting a particular filter for a pair of hosts. Therefore, filters such as:

```
marsh     lillee    IP/TCP/telnet
decimus   bike      IP/UDP/DNS
```

define only telnet traffic between hosts marsh and lillee, and only DNS traffic between hosts decimus and bike. This would allow users to examine different specific traffic details during one packet capture. However, the current filter structure does not support this format. In relation to the above filters, the current structure would match telnet and DNS packets between marsh and lillee, as well as decimus and bike. Therefore any DNS packets exchanged between marsh and lillee would also be matched. Ideally, the defined filter structure should provide a mechanism which allows host names to be added per filter. Preliminary work carried out by the author has revealed that the entire filter mechanism needs modification. This work is not in the scope of this unit. A proposed solution would be to add a filter node to the filter linked list, where a node has the following format:

```
typedef struct frame {
      int frame_id;              /* Ethernet frame type */
      struct ether_addr shost; /* Source host name */
      struct ether_addr dhost; /* Destination host name */
      struct u_char  subproto; /* Sub protocol */
      struct u_short    sport; /* Source port number */
      struct u_short    dport; /* Destination port number */
      struct u_char port_test; /* Destination port number */
 }
```

Implementing such a structure will reduce the efficiency which is present in the current system, but can provide a more intuitive method of filtering.

Another limitation of the current filtering scheme lies in the deletion of particular filters. Currently filters are set when the Done button of the filter popup is clicked. Before this event occurs, users may set and delete any protocol and host filters using the mechanisms provided. However, once set those filters remain active until the Clear All button is clicked on a subsequent popup, or the All packets filter is chosen. However, the All Packets filter will only capture packets between the latest set host filter pair (i.e. if cain and abel were the last set host pair, then the filter would capture all packets between this pair.). Therefore host filters also need to be removed. Currently, there is no mechanism to remove individual host filter pairs at any time, or individual filters on consecutive popups. Rather, the active list is always displayed thereby allowing the user to determine which filters are active at any one time. This problem can be overcome by developing a method to remove particular nodes of the protocol and host filtering structure. This requires some complex modifications of the filtering mechanism because the current filter structure makes use of sublists to maintain efficiency. The task will be easier if a filter scheme as proposed above is developed.

6 Conclusion

The new filtering mechanism addresses all of the aims and objectives, as outlined in Section 2.4 The new mechanism applies good user interface design principles to filter selection, an area lacking in sophistication in the previous version of *packetman*. The filtering mechanism was designed with a more comprehensive filtering system in mind, and this requires further work to be carried out to fully exploit the features of the user interface.

The proposed complete filtering system can allow users to specify detailed logical expressions which take advantage of the speed and efficiency of operating system level packet captures. The filtering language for this proposed system has already been implemented, and further work in the machine independent representation is required before this system can be exploited.

References

Barron, G. (1993). Pakcetman v1.1. Report for Computing Project 394, November 1993. Project Report.

Benko, G. (1992). LAN based packet analysis for a TCP/IP network on X-Windows. Report for Computing Project 394.

Heckel, P. (1984). *The Elements of Friendly Software Design.* Warner Books, Inc.: New York, 666 Fifth Ave., New York, NY 10103.

McCanne, S. and Jacobson, V. (1992). The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *1993 Winter USENIX conference, January 25-29, 1993, San Diego, CA.*

Schulze, M., Benko, G., and Farrell, C. (1993). Homebrew Network Monitoring: A Prelude to Network Management. Technical Report TR 9-1993, Dept. of Computer Science, Curtin University of Technology, Western Australia, Curtin Univerity of Technology, GPO Box U1987, Perth 6001, WA, Australia.

7 Appendix - Source files