Packet Filtering on the IXP1200 Network Processor

(Project Report)

Harkirat Singh, Kathryn Mohror, Dilip Sundarraj, Satyajit Grover, Gokul Huggahalli
Team 2
TCP/IP Internals, Spring 2003
{harkirat, kathryn, dilip, satyajit, huggahag}@cs.pdx.edu

I. Requirements

To design filters to perform the following:

1) Layer 2 Filter: This filter should classify the packets as Unicast, Multicast or Broadcast and maintain a counter for each.
2) Layer 3 Filter: This filter should classify packets as TCP, UDP, ICMP or OTHER and maintain a counter for each.
3) Complex Filter: This is a complex filter that classifies packets as TCP and UDP on specific ports. The most popular ports like 21, 22, 23, 80 etc. are considered and hard-coded. A counter is maintained for each (protocol, port) tuple.
4) Dynamic Filter: This filter performs complex filtering of TCP and UDP packets based on the user requirements. The user could specify the particular protocol, source IP address, destination IP address, source port and destination port whose packets s/he is interested in. This filter can also look for virtual connections if all four parameters are provided.

II. Design

In this study we use two ACEs namely, Ingress ACE and Filter ACE. Ingress ACE is the default implementation provided with the Intel SDK. We have implemented a custom *Filter ACE*.

Filter ACE has two components, micro and core. The primary function of Filter microace is to increment counters based on valid packets as per a set of rules. Some of these rules are hard-coded and one is user defined in a startup configuration file. The microace raises an exception and passes along packets that match the user defined rule. The core component runs on the SA processor. Its job is to periodically poll the counters and display them accordingly. This poll time is a user defined parameter in the IXP1200 configuration file. In case it is not provided the polling time will default to a hard-coded value. The core also handles the exception raised by the microace when a packet matches the dynamic filter described in the requirements.

The microblock components of both the Ingress and Filter ACEs run on one microengine.

A. Life of the Packet

As soon as the packet is assembled in the SDRAM and the rcv_ready bit is set, the Ingress Ace gets the packet. The Ingress Ace then hands the packet to our Filter Ace which performs different checks on the packet by scanning its header.

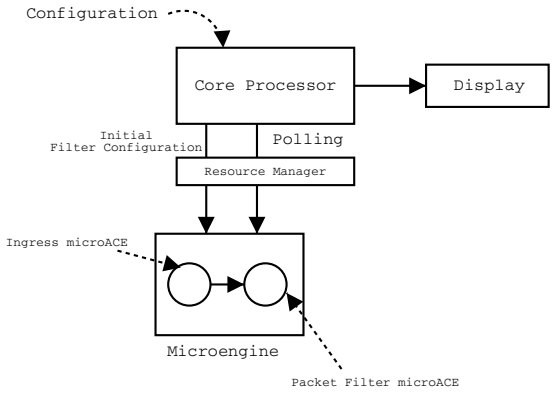

Fig. 1. Filter architecture

The Layer 2 filtering is performed by examining the ethernet header. The source and destination ethernet addresses are read and compared to broadcast, multicast, and unicast address families. According to the type of packet, the corresponding counter is incremented. Following this, the Layer 3 filtering is performed by scanning the IP header of the packet. The type of the packet is determined as TCP, UDP, ICMP or OTHER and the corresponding counter is incremented. The Complex filter looks into the transport header that contains the port numbers of the application that is the recipient of this packet. Separate counters for each port for both TCP and UDP are maintained.

For the Dynamic Filter, the user inputs the protocol, source IP address, destination IP address, source port and destination port which need to be monitored for packets. Packets matching this filter are sent to the core component in the SA through the use of an exception. If the user specifies a 0 for any of these four parameters we take it to mean *any value* for that parameter. In that case, the four tuple is incomplete and cannot be classified as a virtual connection. We can, however, match packets based on that rule. On the other hand, if the entire four tuple is specified, a counter is maintained for each direction of the virtual connection. In both cases, the dynamic filter specification is read from an external file by the initialization code in the core component. The core component of our Filter Ace then stores the filter parameters into a location in scratch-pad memory in the structure shown in Figure 2. The micro component of the ace reads this memory location for the

```
struct connection {
   uint32_t  type;
   uint32_t  src;
   uint32_t  dst;
   uint32_t  src_port;
   uint32_t  dst_port;
}
```

Fig. 2. Structure for holding dynamic filter

parameters and checks the header of the packet against these values.

All the filters are programmed as a single microace that executes the static filters on every execution. The choice on whether to run the dynamic filter is based on the user's requirements. If the user configuration file does not specify a TCP or UDP filter, the dynamic filter is not executed. Separate counters are maintained for each and are incremented when a matching packet is identified.

The counters that are maintained by the micro component of the Filter ACE are accessible by the Strong Arm. The SA periodically reads the values in each of the counters and displays them. The Strong Arm processor checks these counters once every few seconds and displays them to the terminal. The number of seconds can be provided as a parameter in the IXP1200 configuration file.

III. Data Structures

We have defined two data structures that would help us maintain counters for different kinds of packets captured by the different filters. The stats structure (Figure 3) contains the different counters and it is accessible by both the core component and the microblock component of our filter ace.

The stats structure was adopted from Andrew's code and modified. The *counter* field is used to maintain the counter for the number of packets on the virtual connection between the (source IP, source port) and (destination IP, destination port) as specified by the user. The *vc* counter maintains the count of packets going the other way on this virtual connection. The user specifies the connection four-tuple using our dump program that creates the connection structure as shown in Figure 2. The connection structure has a type field to store the type of protocol, which can be tcp or udp. The src, dst fields are the source and destination IP addresses. The src_port and dst_port are the source and destination ports. If any of these fields are not specified explicitly then it means that the user does not care about that particular field and it could take any value. In order to achieve this, we set the default value of these fields to be 0, in which case we do not compare for a match on that particular field.

IV. Algorithm

According to our design architecture we have designed our system to apply the Layer 2, Layer 3 and Complex filter for all the packets that are being received on our interface 1 of the IXP card.

```
typedef struct _stats {
uint32_t   len;
struct bpf_insn  instrns[25];
   uint32_t   counter;
   uint32_t   unmatched;
   uint32_t   type;
   uint32_t   bcast;
   uint32_t   mcast;
   uint32_t   ucast;
   uint32_t   icmp;
   uint32_t   tcp;
   uint32_t   udp;
   uint32_t   other;
   uint32_t   tcpssh;
   uint32_t   ipOptions;
   uint32_t   tcpftp;
   uint32_t   tcphttp;
   uint32_t   tcptelnet;
   uint32_t   udpdns;
   uint32_t   udprip;
   uint32_t   vc;
   struct connection conn;
}stats_t;
```

Fig. 3. Structure for holding counters and an instance of the dynamic filter

1) We initially get a pointer to the packet that was currently captured into one of the General Purpose Registers.
2) The MAC destination address is then extracted and tested to see if it is a broadcast or multicast address. For broadcast, we check if the address is 0xffffffff. If so, we increment the broadcast and multicast counters since broadcast is also a type of multicast. For multicast, we check if the least significant bit of the first byte of the MAC Destination address is 1. If it is, we increment the multicast counter. If these tests fail then we increment the unicast counter.
3) Next we check for the type of packet in the MAC header. If it is an IP packet then we continue further processing; otherwise we skip and process the next packet.
4) If we find that it is an IP packet then we extract the header length from the header length field in the IP header.
5) We check if the header length is greater than 20 bytes. If it is, then the IP header has options and so we increment a counter for packets received with IP options. Then, we branch to the IpOptions section of the code, where we extract source and destination ports from the transport layer header at the appropriate offsets. Otherwise, we extract them from the known locations, i.e. fields following the IP header and in the beginning of the transport layer header.
6) We compare the protocol type in the IP header protocol field with ICMP, TCP or UDP. We increment the appropriate counters accordingly. If none of these match then we increment the OTHER counter.
7) If we find the transport layer protocol is TCP or UDP

then we match it against the type member of the connection structure supplied by the user. If match is successful then we set the conn_filter flag and extract the IP source and destination addresses from the IP header.

8) We also check for the popular TCP ports like SSH (22), FTP (21), TELNET (23), HTTP (80) and UDP ports like DNS (53), RIP (520). We increment the appropriate counters in the stats structure.
9) Next, we check for the conn_filter flag (as set in Step 7). If it is set and none of the parameters of the connection is 0, then we need to trace the virtual connection. We have separate counters for each direction of the connection. If any parameter is set to 0 then by our design we do not consider it a virtual connection. In this case we take it to mean *any value* for that parameter. If the packet satisfies the other non-zero given parameters we increment just one counter. If both the cases fail then we increment the Unmatched counter.
10) Finally, we send an exception to the core component of the Filter Ace in case the packet matches our dynamic filter and also pass the packet up to it. [1]

V. Implementation and Results

We implemented the two components of our Filter ACE as described in Section II. The core component was written in C. The microace implementing our filtering algorithm was implemented in IXP1200 microcode. Our design involved the generation of exceptions for packets that matched our dynamic filter. The SA core would then decide what needs to be done with the exception raising packet. This feature was implemented and was found to work. However, we observed that when the traffic in the network for packets matching the dynamic filter is high, our filter stops working due to the sheer load of exceptions that are being generated. This could be due to the SA's inability to handle a very large number of exceptions. We performed a number of tests on our code. These test cases and their results are outlined in a separate Test Plan document.

VI. Acknowledgements

We would like to thank the class TA, Andrew Jauri, for his help with the IXP machines and the tutorials.

References

[1] Intel, *Intel IXA SDK Programming Framework*. Intel Press.
[2] E. J. Johnson and A. R. Kunze, *IXP-1200 Programming*. Intel Press, March 2002.
[3] G. R. R. Wright and W. R. Stevens, *TCP/IP Illustrated: The Implementation, Vol. 2*. Addison Wesley, October 1994.
[4] D. Comer, *Network System Design Using Network Processors*. Prentice Hall, February 2003.

[1] This feature was implemented, but commented out. See Section V for details.