*

# Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages

*Masanobu Yuhara*
Fujitsu Laboratories Ltd.
1015 Kamikodanaka
Nakahara-ku
Kawasaki 211, Japan

*Brian N. Bershad*
Department of Computer Science
and Engineering FR-35
University of Washington
Seattle, WA 98195

*Chris Maeda*
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

*J. Eliot B. Moss*
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

**Abstract**

This paper describes a new packet filter mechanism that efficiently dispatches incoming network packets to one of multiple endpoints, for example address spaces. Earlier packet filter systems iteratively applied each installed filter against every incoming packet, resulting in high processing overhead whenever multiple filters existed. Our new packet filter provides an associative match function that enables similar but not identical filters to be combined together into a single filter. The filter mechanism, which we call the Mach Packet Filter (MPF), has been implemented for the Mach 3.0 operating system and is being used to support endpoint-based protocol processing, whereby each address space implements its own suite of network protocols. With large numbers of registered endpoints, MPF outperforms the earlier BSD Packet Filter (BPF) by over a factor of four. MPF also allows a filter program to dispatch fragmented packets, which was quite difficult with previous filter mechanisms.

## 1 Introduction

In this paper, we describe a new packet filter mechanism, which we call MPF (Mach Packet Filter), that efficiently handles many filters that are active at the same time. Our new packet filter also supports context-dependent demultiplexing, which is necessary when receiving multiple packets in a large fragmented message. We have implemented our new packet filter in the context of the Mach 3.0 operating system [Accetta et al. 86]. The new packet filter improves performance by taking advantage of the similarity between filter programs that occurs when performing endpoint-based protocol processing. With 10 TCP/IP sessions, MPF is almost eight times faster than the CMU/Stanford packet filter (CSPF) [Mogul et al. 87], and over four times faster than the BSD packet filter (BPF) [McCanne and Jacobson 93]. MPF's performance advantage grows with the number of sessions.

---

*This research was sponsored in part by The Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035, by the Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F 19628-91-C-0168, by Fujitsu Laboratories Ltd., the Xerox Corporation, and Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, Fujitsu Laboratories, Xerox Corporation, Digital Equipment Corporation, the National Science Foundation, or the U.S. Government. Bershad performed this work while at Carnegie Mellon University. Authors' email addresses are {yuhara,bershad,cmaeda,moss}@cs.cmu.edu.

The original packet filters (CSPF and BPF) shared two primary goals: protocol independence and generality. The filters did not depend on any protocol, and future protocols could be accommodated without changing the kernel. MPF shares these two goals, as it is implemented as an extension to the base BPF language. Consequently, a packet filter program built for BPF will work with our system. Although MPF has been implemented for the Mach operating system, it requires no changes to the Mach microkernel interface, and has no Mach-specific aspects. Other BPF implementations could be extended to support MPF programs, and our implementation should port easily to other operating systems that support packet filters.

## 1.1 Motivation

A packet filter is a small body of code installed by user programs at or close to a network interrupt handler of an operating system kernel. It is intended to carry an incoming packet up to its next logical level of demultiplexing through a user-level process. An operating system kernel implements an interpreter that applies installed filters against incoming network packets in their order of arrival. If the filter accepts the packet, the kernel sends it to its recipient address space. Two packet filters, CSPF and BPF, are common in today's systems. CSPF is based on a stack machine. A CSPF filter program can push data from an input packet, execute ALU functions, branch forward, and accept or reject a packet. BPF is a more recent packet filter mechanism which, instead of being stack-based, is register-based. BPF programs can access two registers (`A` and `X`), an input packet (`P[]`), and a scratch memory (`M[]`). They execute load, store, ALU, and branch instructions, as well as a return instruction that can specify the size of the packet to be delivered to the target endpoint. BPF admits a somewhat more efficient interpreter than CSPF [McCanne and Jacobson 93].

With a microkernel, where traditional operating system services such as protocol processing are implemented outside the kernel, the original packet filter provided a convenient mechanism to route packets from the kernel to a dedicated protocol server. Scalability was not important because relatively few packet filters would ever be installed on a machine (typically two: one to recognize IP traffic and one to recognize all other traffic). Unfortunately, a single point of primary dispatch for all network traffic resulted in communication overhead for microkernel-based systems substantially larger than for monolithic systems, in which the protocols are implemented in the kernel [Maeda and Bershad 92]. To address this problem, we have decomposed the protocol service architecture so that each application is responsible for its own protocol processing [Maeda and Bershad 93]. That is, every address space contains, for example, a complete TCP/IP stack. Figure 1 illustrates the structural differences between the two different protocol strategies.

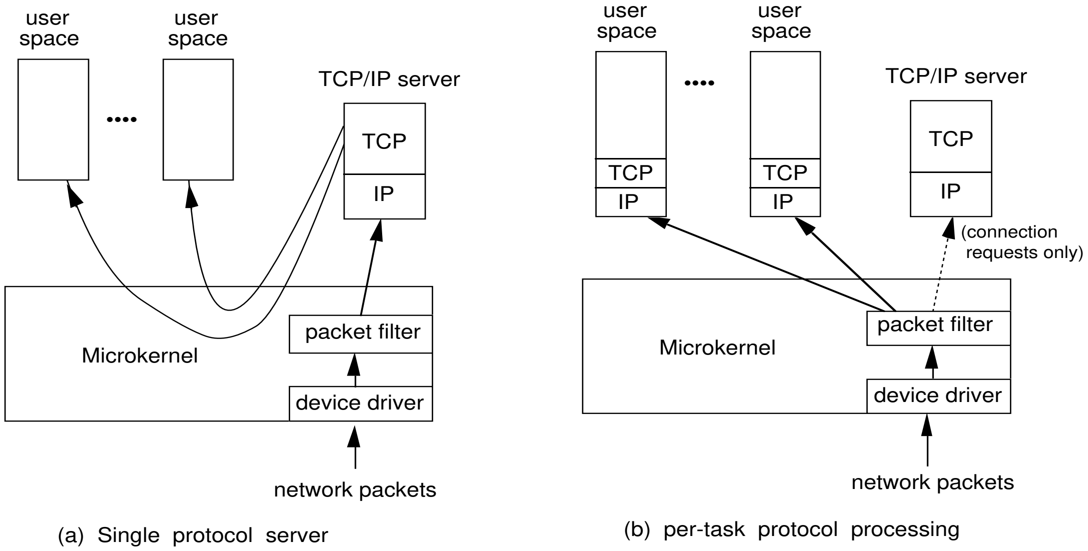


Figure 1: *Two ways to structure protocol processing. In the system on the left, all packets are routed through a central server and then on to their eventual destination. In the system on the right, the kernel routes an incoming, but unprocessed network packet directly to the address space for which the packet is ultimately intended, resulting in lower latency and higher throughput. A central server handles operations without critical performance requirements, such as connection establishment.*

At its core, our new protocol architecture relies on the kernel's packet filter mechanism to deliver incoming packets to the appropriate address space. Our application-level protocol processing architecture revealed two serious problems with existing implementations of the packet filter:

1. *The packet filter is not scalable.* The dispatch overhead grew linearly with the number of potential endpoints. For even a workstation-class machine, it is not uncommon to have several hundred protocol endpoints in use at a time, so scalability becomes critical for efficient demultiplexing.

2. *A packet filter is unable to efficiently recognize and dispatch multipacket messages.* Some protocols require information in the previous or future packets to dispatch a packet. For example, the IP protocol splits one large IP packet into several small IP packets when the underlying data link layer cannot accept a large packet [RFC791]. Moreover, the fragmented packets may arrive out of order. The existing packet filters have no mechanisms for efficiently dealing with fragmentation, let alone out of order delivery. Therefore, they cannot dispatch fragmented packets to any of multiple endpoints. Instead, fragments must all be sent to a higher-level intermediary process using the "packet filter of last resort" at the expense of substantially more kernel messages and boundary crossings.

We have solved these two problems by extending the existing BPF instruction set with new instructions that enable the packet filter implementation to support efficiently large numbers of endpoints and fragmented packets.

To deal with the scalability problem, MPF takes advantage of the structural and logical similarity within a protocol, and attempts to dispatch all packets destined for that protocol in a single step. Typically, filter programs for a particular protocol consist of two parts: one that identifies the protocol and one that identifies the session in that protocol. (The code in Appendix A shows an example BPF program for TCP/IP dispatching.) The first part is exactly the same for all sessions within a protocol, while the second part differs only in the constant values that identify the particular session instance. With MPF, the kernel's filter module internally transforms, or *collapses*, filter programs for the same protocol into a single filter program. Figure 2 contrasts MPF with previous packet filter mechanisms, which execute similar code repeatedly for each protocol session.

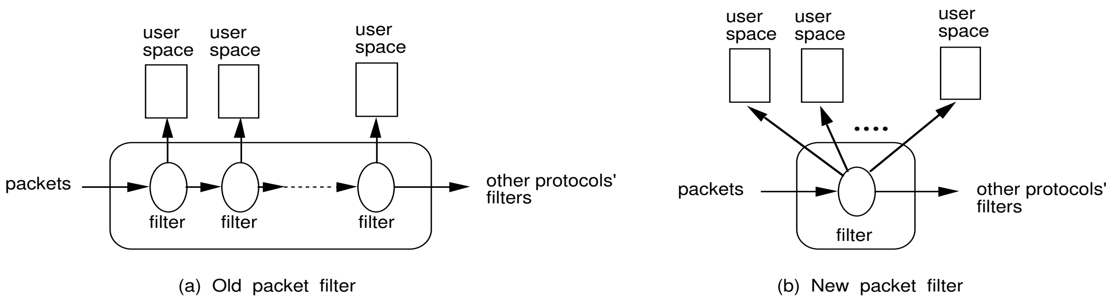


Figure 2: *Redundant (BPF and CSPF) vs. one-step filtering (MPF) for incoming packets*

To deal with the fragmentation problem, MPF provides per-filter state that persists across the arrival of packets. Filter programs can record dispatch information using an early packet, and later retrieve that recorded information to dispatch a subsequent packet in a multipacket message.

### 1.2 The rest of this paper

The rest of this paper is organized as follows. In Section 2 we describe the design and implementation of our single-pass packet filter. In Section 3 we present our technique for dispatching fragmented messages. In Section 4 we discuss the performance of our new packet filter mechanism. In Section 5 we present our conclusions.

## 2 Fast dispatch of incoming packets to multiple endpoints

As mentioned in the previous section, filters generally perform two levels of dispatch. The first level dispatches to a protocol, while the second level to an endpoint within that protocol. The logic for the first level of dispatch is identical for all packets destined to a particular protocol, while that for the second relies on mapping from some number of fields in the packet header to an actual address space/endpoint. For example, in the MPF implementation, the endpoint is identified by a Mach IPC port [Draves 90], which is explicitly defined when the filter is installed.

We have introduced an associative match instruction (`ret_match_imm`) that allows MPF to exploit the fact that a packet is dispatched first to a protocol and then to a session within that protocol. The `ret_match_imm` instruction, described in Table 1, is a combination of the original packet filter's compare and return instructions.

| MPF match sequence | Equivalent BPF match sequence |
|---|---|
| <pre>ret_match_imm #3, #ALL<br>key #key0<br>key #key1<br>key #key2</pre> | <pre>        ld M[0]                 ; A = M[0]<br>        jeq #key0, k1, fail     ; if (A == key0) goto k1; else goto fail;<br>k1:     ld M[1]                 ; A = M[1]<br>        jeq #key1, k2, fail     ; if (A == key1) goto k2; else goto fail;<br>k2:     ld M[2]                 ; A = M[2]<br>        jeq #key2, ok, fail     ; if (A == key2) goto ok; else goto fail;<br>ok:     ret #ALL                ; return the whole packet<br>fail:   ret #0                  ; abort this filter</pre> |

Table 1: *The* `ret_match_imm` *instruction from MPF and its equivalent sequence from BPF. The first argument of* `ret_match_imm` *indicates the number of data items to be compared. The subsequent* `key` *pseudo instructions provide immediate data. These immediate values are compared with the values in the scratch memory: M[0], M[1], M[2], respectively. If the corresponding values are equal, then the filter returns with success. The second argument of the* `ret_match_imm` *instruction specifies the number of bytes of the packet sent to the recipient ("ALL" indicates the entire packet). If any pair of the corresponding values is not equal, the filter terminates with failure, and the packet is not sent to the recipient for this filter.*

The MPF implementation uses the `ret_match_imm` instruction to collapse multiple filter programs into one by converting the `ret_match_imm` instruction into a fast associative lookup that precedes the dispatch. When a user task installs a new packet filter, the kernel's filter module scans the the program for an associative match instruction. If one is found, the kernel then searches for a previously installed filter program having identical code, but differing only in the immediate values contained in the `key` pseudo-opcodes following the associative match instruction.

Each suite of similar filters has a hash table in which each entry contains the keys and Mach IPC port of one of the suite's component filters. Upon finding a similar filter, the kernel creates an entry for the new filter in the hash table that corresponds to the similar filter. The kernel then discards the new filter, as it has been effectively collapsed into an existing one. If no match instruction is found, or no similar filter program exists, then the filter is installed with no optimizations applied.

Figure 3 illustrates a sample MPF program that reveals the split-level dispatch and use of the new instruction. The code sequence performs the same function as that in Appendix A. The filter accepts packets sent to a TCP/IP session specified by source IP address (`src_addr`), source TCP port (`src_port`), and destination TCP port (`dst_port`). The combination of these constant parameters is unique for a particular session. Other constant parameters (including the destination IP address, `dst_addr`) are the same for all TCP/IP sessions. The first part (A) of the MPF program checks if the packet uses the TCP/IP protocol. The second part (B) extracts the TCP session information from the packet and puts it into the scratch memory. Parts (A) and (B) are common to all TCP/IP filters. The last part (C) determines if the packet is in fact destined for this particular filter (session).

When a packet arrives and the kernel's filter mechanism processes a collapsed filter, it executes the common part (A and B from the example program) just like a conventional program. If the the common part rejects the packet, it is rejected by the whole filter suite, avoiding redundant execution of the filter's common sequence. If the packet is accepted in the common part, the filter module executes the unique part, namely the `ret_match_imm` instruction. Using the values in the scratch memory (M[0] .. M[2] in the example), the kernel searches the filter's hash table for a match. If the search is successful, then the packet is sent to the corresponding receive port. If the search fails, then the collapsed filter suite rejects the packet (but other filter programs might still apply).

The approach described in this section is powerful and easy to implement, but extremely restrictive. An MPF filter program can have only one common sequence, the filter must begin with the sequence, and no instructions may follow the associative match that marks the end of the sequence. Clearly, more general alternatives could be used. For example, a filter program could have an arbitrary number of common sequences occurring at any location, and any degree of processing outside the common parts could be permitted. Our requirements, though (fast endpoint demultiplexing), combined with the two-level dispatch common to most protocols, suggested a solution that was simple to implement and right most of the time, rather than one that was much more complicated to implement and right about as often.

```
/* Part (A) */
        begin                           ; MPF/BPF identifier
        ldh     P[OFF_ETHERTYPE]        ; A = ethertype
        jeq     #ETHERTYPE_IP, L1, Fail ; If not IP, fail.
L1:
        ld      P[OFF_DST_IP]           ; A = dst IP address
        jeq     #dst_addr, L2, Fail     ; If not from dst_addr, fail.
L2:
        ldb     P[OFF_PROTO]            ; A = protocol
        jeq     #IPPROTO_TCP, L3, Fail  ; If not TCP, fail.
L3:
        ldh     P[OFF_FRAG]             ; A = Frag_flags|Frag_offset
        jset    #!Dont_Frag_Bit, Fail, L4
                                        ; If fragmented, fail.
L4:

/* Part (B) */
        ld      P[OFF_SRC_IP]           ; A = src IP address
        st      M[0]                    ; M[0] = A

        ldxb    4 * (P[OFF_IHL] & 0xf)  ; X = offset to TCP header

        ldh     P[x + OFF_SRC_PORT]     ; A = src TCP port
        st      M[1]                    ; M[1] = A

        ldh     P[x + OFF_DST_PORT]     ; A = dst TCP port
        st      M[2]                    ; M[2] = A

/* Part (C) */
        ret_match_imm #3, #ALL          ; Compare keys and M[0..2].
        key     #src_addr               ; If matched, accept the
        key     #src_port               ; whole packet. If not,
        key     #dst_port               ; reject the packet.
Fail:
        ret     #0
```

Figure 3: *An MPF program for a TCP/IP session.*

# 3 Dispatching fragmented messages

Fragmentation occurs when a lower-level protocol layer cannot transfer the entire packet of a higher-level protocol. For example, consider the case of a protocol stack consisting of Ethernet, IP, and UDP. Since UDP messages can be larger than the maximum Ethernet message (4k bytes or larger for NFS packets over UDP, but only about 1500 bytes for Ethernet), they must be sent as a number of IP fragments and reassembled at the destination host. Each IP fragment contains a common message id that is unique to the UDP message, offset and length information that describe which part of the UDP message the fragment contains, and finally the data. Only the first fragment contains the UDP header, which includes the UDP source and destination port numbers. The message id, offset, and length are used by IP to reassemble the incoming UDP message.

Demultiplexing incoming fragments is difficult for several reasons: only the first fragment contains the transport protocol header which provides the information needed to determine the target endpoint, fragments may arrive out of order, and some fragments may not arrive at all. We wanted to support simple and efficient demultiplexing of fragments using the packet filter. We were not concerned with performing actual reassembly at the packet filter layer, as we expected that service to be provided by user-level code.

To deal with fragmentation, we introduced a per-filter memory that allows packet filters to link dispatch information present in only the first fragment of a message (for example, the UDP port numbers), and information present in all fragments (for example, the unique message id).[1] Filter programs can record the higher level session dispatch information and associate it with a lower level message id, allowing them to dispatch fragments to the correct endpoint. This association persists for a finite time, after which it is automatically removed.

Because fragments don't always arrive in order, we also allow a filter to postpone processing of a fragment when the first fragment of a message is not the first to arrive. Since no dispatch information is available for the earliest arriving fragments, these fragments are postponed, and processed only after other packets have arrived. Hopefully, the

[1] We assume that the fragments of a message between any pair of source and destination addresses share a message id that is unique to the message.

dispatch information will have become available. Postponed packets are dropped if it appears the dispatch information will not become available, or if space to record the postponed packets is exhausted.

### 3.1 Details of the approach

We have added four new instructions to the packet filter instruction set to handle fragmentation: `register_data`, `ret_match_data`, `jmp_match_imm`, and `postpone`. These instructions are described in Table 2, and demonstrated in Appendix B which shows the flow of a UDP/IP filter program that handles fragmented packets.

| **Instruction** | **Description** |
|---|---|
| `jmp_match_imm #N, Lt, Lf` | This instruction is used to identify the first of a fragmented packet. The instruction is similar to the `ret_match_imm` instruction in that the N immediate data values following the instruction are compared with M[0] .. M[N-1] of the scratch memory. This instruction conditionally jumps forward depending on the result of the comparison. If the data match, control transfers to `Lt`, otherwise control transfers to `Lf`. |
| `register_data #N, #T` | This instruction is used to bind an endpoint to a message id. The scratch memory values M[0]...M[N-1], intended to be the message id, recorded in the filter's static memory. These values are removed after T milliseconds. T should be longer than the expected time to receive all fragments in a message. This instruction may only execute following a successful `jmp_match_imm`, which creates the higher association between a particular named endpoint and the first fragment destined for that endpoint. |
| `ret_match_data #N, #R` | This instruction is used to return a fragment in a large message to the appropriate endpoint. The instruction compares M[0]..M[N-1] of the scratch memory, intended to contain the message id of the current packet, with the static memory of this filter. If the values are the same, R bytes of the packet are sent to the recipient of this filter. If not (or if the static memory values do not exist), execution continues with the next instruction. |
| `postpone #T` | This instruction postpones processing of the current packet, deferring it to some later time. If a postponed packet is chosen for processing, it may be postponed again. The packet is discarded after T milliseconds from its original arrival, although it may be discarded earlier because of storage limitations. |

Table 2: *New instructions to support handling of fragmented packets. All but the* `jmp_match_imm` *instruction are expected in the common part of a filter, enabling the collapse of filters that dispatch fragments.*

The `jmp_match_imm` instruction is a branching version of the `ret_match_imm` instruction described in the previous section. If the match fails, the program branches to the false-case label. If the match succeeds, the program branches to the true-case label. As a side effect of matching, the receive port associated with the key data following the `jmp_match_imm` instruction becomes associated with the currently running packet filter. If the filter then executes a normal `return` instruction, the associated receive port is recalled and used as the recipient. In this way, we avoid having to explicitly manipulate kernel descriptors (really, IPC ports) within the packet filter, yet are able to collapse filters that handle fragmentation.

The `register_data` and `ret_match_data` instructions store and retrieve the fragmentation information. When a packet filter executes the `register_data` instruction, the contents of its scratch memory are used as keys associated with its receive port in a second (filter-specific) hash table. The `ret_match_data` instruction uses this hash table to provide fast lookup on the fragment information. Each entry in the second hash table has its own expiration time specified by the filter program.

The `postpone` instruction addresses the situation where a later fragment arrives before the first fragment. We assume that out-of-order arrival is rare, and use a simple postponement mechanism. A postponed packet is placed on a pending packet queue. Pending packets are reprocessed immediately after each new packet is filtered. Of course, the filter program may postpone the packet again. However, the packet's expiration time is set when it is first postponed, and the packet will be dropped after that time, or if the number of postponed packets becomes too large.

The fragmentation support described in this section imposes no overhead for filtering non-fragmented packets. Note that the kernel does not do reassembly; that work occurs in the endpoint address space itself. The kernel merely assists in routing the fragments to the appropriate reassembly routine.

# 4 Performance

In this section we discuss the performance of our new packet filter mechanism by comparing it to BPF and CSPF. We answer four questions about performance:

- how does filter processing overhead grow as a function of the number of installed protocol endpoints?
- what is the round-trip latency for messages received through the packet filter as a function of the number of installed protocol endpoints?
- what is the overhead to install a new filter?
- how long does it take to detect and handle a fragmented message?

We conducted our measurements on a DECstation 5000/200 (25 MHz MIPS R3000, 64 KB instruction cache, 64 KB data cache, Lance AMD7990 Ethernet controller) running Mach 3.0 (MK82) and CMU's Unix single-server server (UX41). All timing measurements were taken using a memory-mapped 25 MHz free-running counter.

## 4.1 Packet filter latency

The most important performance metric for our new packet filter mechanism is its scalability, which we measure in terms of filtering latency as a function of the number of active sessions. By latency, we mean the time required for the packet filter module to determine which filter accepts an incoming packet. This time is constant for all packet sizes, since the filters examine only the packet headers. Latency does not include the time spent in the kernel's interrupt handler, message transmission time, or the time to dispatch a matched packet to an address space; these times are constant for all the packet filter mechanisms. To put the numbers in this section in context, Table 3 shows the time required for the Ethernet device driver to service an incoming packet, and for the kernel to deliver it to the destination address space. These operations occur before and after the packet filter runs, respectively. Packet filtering (demultiplexing) should take much less time than these operations, which are dominated by data movement.

| Operation | Packet Size | |
|---|---|---|
| | 64 | 1514 |
| Time to read an incoming packet from the Ethernet device. | 0.1 | 0.5 |
| Time to move an incoming packet from the kernel to the destination address space. | 0.1 | 0.2 |

Table 3: *The time (in milliseconds) required for the device driver to service incoming packets and for the kernel to deliver an incoming packet to its destination address space. These times were measured on a DECstation 5000/200 running Mach 3.0.*

Our latency tests reflect just that overhead required to interpret and initiate a packet dispatch to the receiver's address space. To isolate cache effects, we ran the controlled benchmark with cold and warm caches. Worst-case (cold-cache) measurements were obtained by flushing the caches before applying a filter. Best-case (warm-cache) measurements were obtained by running the benchmark without flushing the caches. Actual performance will vary between these extremes, with the operating point depending on the frequency of packet arrival, and the nature of other system activity.

We varied the number of TCP/IP filters installed in the filter module (no other protocols were registered), and measured the latency for the three packet filter implementations. For MPF, all filters could be collapsed into a single filter. Figure 4 shows the results. The latency of both BPF and CSPF grows linearly as the number of sessions increases because both filter mechanisms must run a filter program for each session. The latency for MPF, on the other hand, is insensitive to the number of sessions since only one filter program is executed to demultiplex packets for all sessions. With only ten TCP sessions, MPF (0.035 ms) shows performance that is 7.8 times better than CSPF (0.273 ms) and 4.3 times better than BPF (0.152 ms) in the warm cache case. At 100 sessions, the total demultiplexing time for either BPF or CSPF (add the times in Table 3 to the filtering time) is on the order of the round trip time for many finely tuned network protocol implementations [Schroeder and Burrows 90].

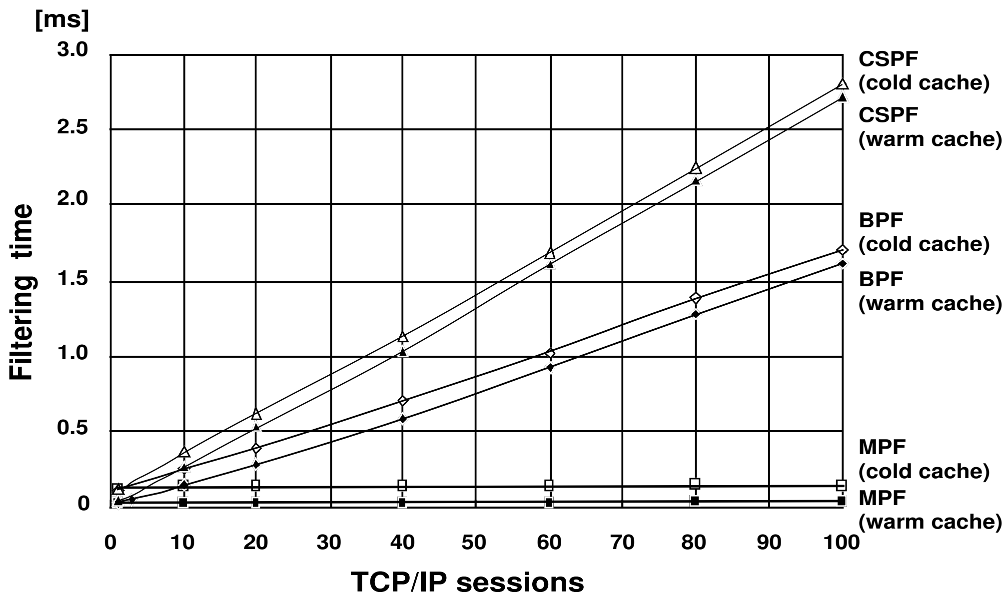


Figure 4: *Filtering time of the three filter mechanisms.*

## 4.2 Round-trip times

To observe the impact on end-to-end application performance, we measured the round-trip time of UDP/IP packets when the protocol service is implemented in the application's address space [Maeda and Bershad 93]. We ran two DECstation 5000/200's in single user mode, connected via private Ethernet. One of the hosts sent UDP packets with one data byte to the other, which replied with the same message. Each host installed the same number of filters. Packets were sent repeatedly, therefore this experiment roughly corresponds to the warm-cache case in Figure 4.

The Mach 3.0 kernel dynamically reorders packet filters so that more active filters are processed before less active ones. We disabled filter reordering so that the filter for the active session was always processed last. This gives us a worst-case estimate of round-trip latency for CSPF and BPF, rather than the best-case that would be given by reordering. In practice, the actual latency will lie somewhere between the best and the worst case, and will be determined by traffic patterns. In contrast, MPF latency is independent of the traffic patterns, and has the same best-case and worst-case latency.

Figure 5 shows the round-trip time of a UDP packet as observed by the application program. The time includes protocol stack processing, interprocess communication within a host, and physical network transfer, as well as packet filtering. MPF is 12% faster than BPF with only 10 sessions, and is over four times faster than BPF with 100 sessions.

In contrast to the filtering latency itself (Figure 4), the round-trip time for UDP using MPF grows slightly as more filters are added. This is due to the combination of our experimental methodology, and our implementation of the Internet protocols. We create 100 filters by creating 100 Unix processes, each with its own UDP endpoint, and each with its own version of an Internet protocol library (UDP/IP and TCP/IP). The library includes several periodic threads that ran during the experiment. These threads create an artificial load on the machine, increasing the wakeup latency of the thread that performs the actual protocol processing, thereby increasing the round-trip time.[2] CSPF and BPF are similarly affected but by no more than the slope indicated with MPF.

## 4.3 Filter installation overhead

As described in Section 2, the MPF module must analyze a filter program during installation. Consequently, the overhead of installing a filter under MPF might be larger than that for BPF or CSPF, which do not collapse similar filters. Two aspects of installation are important: the time to install a new filter (session) that uses a protocol for which a previous filter has already been installed and the time to install a new filter that uses a protocol for which no previous filter has been installed.

[2] This background activity is a bug in the library implementation, which is being fixed for the next release.

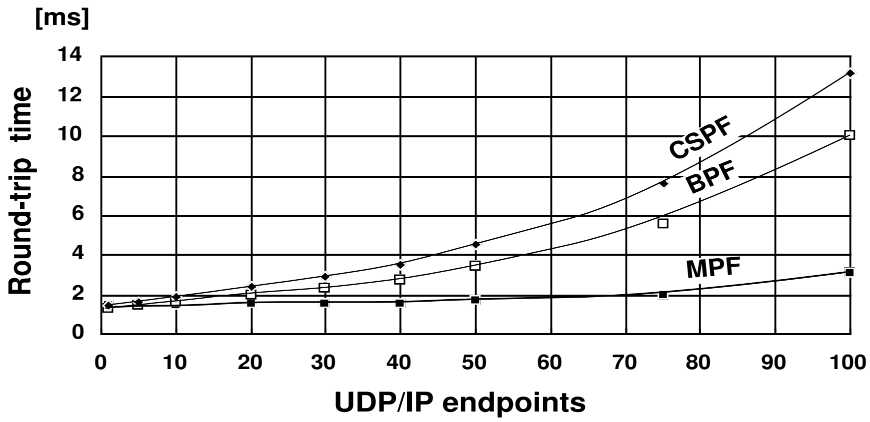


Figure 5: *UDP/IP round-trip time as observed by an application program as a function of the number of installed endpoints.*

We measured filter installation overhead by installing, one-by-one, new filters that were all sessions of a single protocol (first case, Figure 6), or that were all sessions of different protocols (second case, Figure 7). Both cases show the installation time to be comparable. In the first case (many filters, same protocol), when the number of installed programs is small, BPF and CSPF filter installation time is about twice as fast as MPF, because MPF must compare the new filter program against each installed program. As the number of filters increases, the performance difference diminishes; with 75 sessions, the installation time is about the same for the three filter mechanisms. All filter mechanisms must search previously installed filters to determine whether a new filter is being installed, or an existing one replaced. The use of a more efficient internal representation (hash table vs. linked list) in MPF allows the installation time to grow more slowly.

In the second case (many filters, different protocols) the installation time for MPF grows linearly with the number of filters while the installation time for BPF and CSPF remains constant. The increase is because MPF tries to match each new filter with each existing filter, fails, and then prepares a separate hash table for the new filter. In practice, though, we expect that the number of protocols (and so the number of uncollapsed filter programs) to be small.

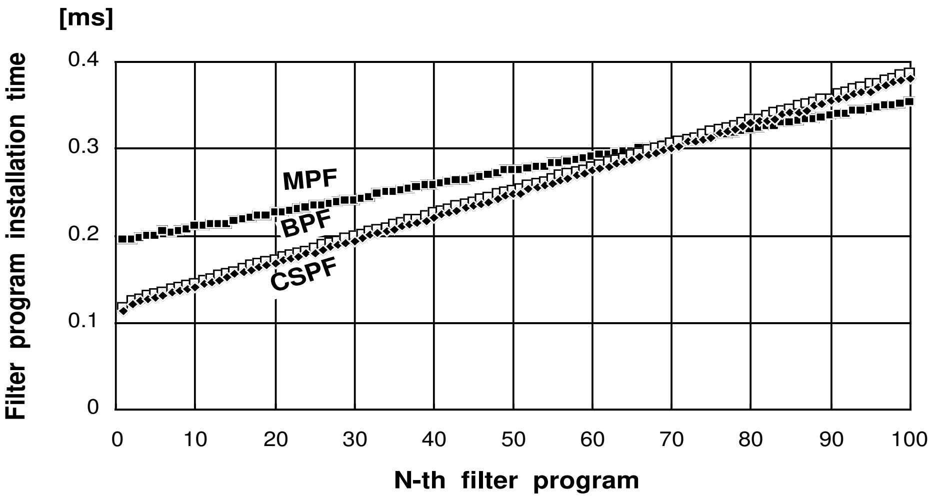


Figure 6: *Filter program installation time. All filters are for for the same protocol.*

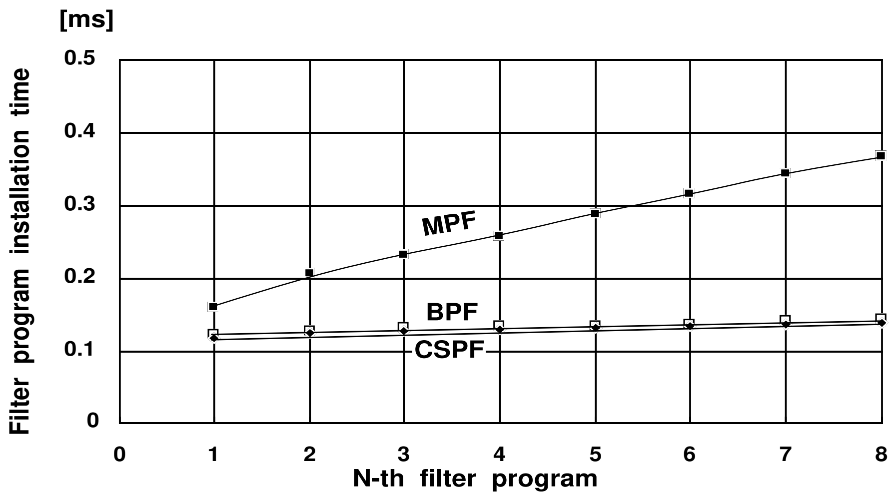


Figure 7: *Filter program installation time. Each filter is for a different protocol.*

## 4.4 Fragmentation overhead

Since neither CSPF nor BPF supports fragmentation, protocol implementations that use them must install "default" packet filters to route message fragments to a dedicated server, where they are reassembled and forwarded to their final destinations. In contrast, MPF provides support to route fragments directly to their final destinations, eliminating the intermediate processing in the server.

Table 4 shows filtering latency for fragmented UDP/IP packets using the MPF program shown in Appendix B. When message fragments arrive in order, filtering time for the second and later fragments is no different from filtering time of a non-fragmented packet. For the first fragment, though, MPF needs more time to register the association between the message id and the destination endpoint. When message fragments arrive out of order, the fragments are processed and postponed until they can be dispatched to their final destination. When the first fragment arrives, the total time to process it and any pending fragments is the sum of the filtering time required for those packets when they arrive in order.

| Packet | Incoming fragment | Fragment processing | [ms] |
|---|---|---|---|
| Non-fragmented | — | | 0.04 |
| Fragmented (in-order) | 1st | Register, dispatch | 0.07 |
| | 2nd or later | Dispatch | 0.04 |
| Fragmented (out-of-order) | 2nd | Postpone | 0.04 |
| | 1st | Register, dispatch (for the 1st), retry, dispatch (for the 2nd) | 0.11 |

Table 4: *The filtering time of non-fragmented and fragmented UDP/IP packets. Two cases for fragmented packets are shown: a case for in-order arrival (the first fragment arrives, then the second fragment), and a case for out-of-order arrival (the second fragment, and then the first fragment).*

As mentioned, the main advantage of MPF's support for fragmentation is that fragmented messages do not have to be routed through an intermediate server. Using the UDP round-trip time program described earlier and 2048 byte packets (two fragments on an Ethernet), we measured a round-trip time of 11.1 ms. for an MPF-based UDP, 11.8 ms. for one based on BPF, and 11.9 ms. for one based on CSPF. CSPF and BPF have similar performance because it is dominated by the time required to move the data from the kernel to the server for reassembly, and from the server to the application. The 7% improvement for MPF comes from avoiding the indirection through a central server.

## 5 Conclusions

MPF is a new packet filter mechanism that can efficiently dispatch small and large packets even in the presence of many sessions, making it suitable for per-task protocol processing. We have introduced a new match instruction that the packet filter mechanism can use as a hint to collapse similar packet filters into one. Collapsing removes repetitive execution of the same code. MPF also supports new instructions to dispatch fragmented packets. Our implementation of MPF is 7.8 times faster for TCP/IP packet filtering than CSPF, and 4.3 times faster than BPF with only ten registered sessions. The source code for MPF can be obtained through anonymous ftp as part of CMU's Mach 3.0 distribution at *mach.cs.cmu.edu*.

## References

[Accetta et al. 86] Accetta, M.J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M.W., "Mach: A New Kernel Foundation for UNIX Development", Proceedings of the Summer 1986 USENIX Conference, pp.93-113, July 1986.

[Draves 90] Draves, R. "A Revised IPC Interface", Proceedings of the First Mach Workshop, pp. 101-121, October 1990.

[Maeda and Bershad 92] Maeda, C., and Bershad, B.N., "Network Performance for Microkernels", Proceedings of the Third Workshop on Workstation Operating Systems, April 1992.

[Maeda and Bershad 93] Maeda, C., and Bershad, B.N., "Protocol Service Decomposition for High-Performance Networking", The Proceedings of the 14th ACM Symposium on Operating Systems Principles, December 1993.

[McCanne and Jacobson 93] McCanne, S., Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture", Proceedings of the Winter 1993 USENIX Conference, pp.259-269, January 1993.

[Mogul et al. 87] Mogul, J., Rashid, R., and Accetta, M., "The Packet Filter: An Efficient Mechanism for User-level Network Code", Proceedings of the 11th ACM Symposium on Operating Systems Principles, pp.39-51, 1987.

[RFC791] Postel, J. B., "Internet Protocol", Request For Comments 791, September 1981.

[Schroeder and Burrows 90] Schroeder, M. and Burrows, M., "Performance of Firefly RPC", ACM Transactions on Computer Systems (8)1, pp.1-17, February 1990.

Appendix

# A BPF example program

```
/*
 * P[i]: packet data at byte offset i.
 * M[i]: i-th word of the scratch memory.
 * Word = 4 Bytes, Half Word = 2 Bytes, Byte = 1 Byte.
 *
 * dst_addr: IP address of this host
 *           (destination IP address of this session)
 *
 * src_addr: source IP address of this session
 * src_port: source TCP port number of this session
 * dst_port: destination TCP port number of this session
 */

        begin                           ; BPF identifier
        ldh     P[OFF_ETHERTYPE]        ; A = ethertype
        jeq     #ETHERTYPE_IP, L1, Fail ; If not IP, fail.
L1:
        ld      P[OFF_DST_IP]           ; A = dst IP address
        jeq     #dst_addr, L2, Fail     ; If not from dst_addr, fail.
L2:
        ld      P[OFF_SRC_IP]           ; A = src IP address
        jeq     #src_addr, L3, Fail     ; If not from src_addr, fail.
L3:
        ldb     P[OFF_PROTO]            ; A = protocol
        jeq     #IPPROTO_TCP, L4, Fail  ; If not TCP, fail.
L4:
        ldh     P[OFF_FRAG]             ; A = Flags|Frag_offset
        jset    #!Dont_Frag_Bit, Fail, L5
                                        ; If fragmented, fail.
L5:
        ldxb    4 * (P[OFF_IHL] & 0xf)  ; X = offset to TCP header

        ldh     P[x + OFF_SRC_PORT]     ; A = src TCP port
        jeq     #src_port, L6, Fail     ; If not from src_port, fail.
L6:
        ldh     P[x + OFF_DST_PORT]     ; A = dst TCP port
        jeq     #dst_port, Suc, Fail    ; If not to dst_port, fail.
Suc:
        ret #ALL                        ; Accept the whole packet.
Fail:
        ret #0                          ; Reject the packet.
```

# B An example filter program that processes fragmented packets

```
/*
 * P[<x>]: Data <x> in the packet.
 * M[i]: i-th word of the scratch memory.
 */
        begin                           ; BPF identifier
        ldh     P[OFF_ETHERTYPE]        ; A = ethertype
        jeq     #ETHERTYPE_IP, L1, Fail ; If not IP, fail.
L1:
        ldb     P[OFF_PROTO]            ; A = protocol
        jeq     #IPPROTO_UDP, L2, Fail  ; If not UDP, fail.
L2:
        ldh     P[OFF_FRAG]             ; A = Flags|Frag_offset
        jset    #0x1fff, Frag2, L3      ; non-zero Frag_offset?
L3:
        /*
         * Frag_offset is zero:
         * Packet is either a 1st frag
         * or an unfragmented datagram.
         * Now we can check the UDP header.
         */
        ld      P[OFF_DST_IP]           ; A = dst IP address
        st      M[0]                    ; first word of key
```

```
        ldxb    4 * (P[OFF_IHL] & 0xf)  ; X = offset to UDP header
        ldh     P[x + OFF_DST_PORT]     ; A = dst UDP port
        st      M[1]                    ; second word of key

        /*
         * Now that the session key is in
         * the scratch memory, we check to
         * see if packet is a fragment that
         * needs to associate the message ID
         * with an endpoint or an unfragmented
         * message that can simply be delivered.
         */
        ldh     P[OFF_FRAG]             ; A = Flags|Frag_offset
        jset    #0x2000, Frag1, NotFrag ; is More_fragment set?
Frag1:
        /*
         * First fragment:
         * Match on UDP session key then
         * register the Message ID.
         */
        jmp_match_imm #2, RegData, Fail
        key     #dst_addr
        key     #dst_port

RegData:
        /*
         * Register the message ID in
         * the per-filter hash table.
         */
        ldh     P[OFF_IP_ID]            ; A = IP Message ID
        st      M[0]
        ld      P[OFF_SRC_IP]           ; A = IP source address
        st      M[1]
        register_data #2, #timeout      ; Associate key with this filter
        ret     #ALL                    ; Deliver first fragment

NotFrag:
        /*
         * Normal unfragmented datagram.
         */
        ret_match_imm #2, #ALL
        key     #dst_addr
        key     #dst_port

Frag2:
        /*
         * Other fragment:
         * Build a key and look in
         * the per filter hash table.
         */
        ldh     P[OFF_IP_ID]            ; A = IP Message ID
        st      M[0]
        ld      P[OFF_SRC_IP]           ; A = IP source address
        st      M[1]
        ret_match_data #2, #ALL         ; return ALL if match
        postpone #timeout               ; postpone if no match

Fail:
        ret     #0
```