The Packet Filter:
An Efficient Mechanism for User-level Network Code

Jeffrey C. Mogul

Digital Equipment Corporation
Western Research Lab

Richard F. Rashid
Michael J. Accetta

Department of Computer Science
Carnegie-Mellon University

November, 1987

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

Code to implement network protocols can be either inside the kernel of an operating system or in user-level processes. Kernel-resident code is hard to develop, debug, and maintain, but user-level implementations typically incur significant overhead and perform poorly.

The performance of user-level network code depends on the mechanism used to demultiplex received packets. Demultiplexing in a user-level process increases the rate of context switches and system calls, resulting in poor performance. Demultiplexing in the kernel eliminates unnecessary overhead.

This paper describes the *packet filter*, a kernel-resident, protocol-independent packet demultiplexer. Individual user processes have great flexibility in selecting which packets they will receive. Protocol implementations using the packet filter perform quite well, and have been in production use for several years.

This paper, in essentially the same form, was originally published in *Proceedings of the 11th Symposium on Operating Systems Principles*, ACM SIGOPS, Austin, Texas, November 1987.

1. Introduction

It is not always appropriate to implement networking protocols inside the kernel of an operating system. Although kernel-resident network code can often outperform a user-level implementation, it is usually harder to implement and maintain, and much less portable. If optimal performance is not the primary goal of a protocol implementation, one might well prefer to implement it outside the kernel. Unfortunately, in most operating systems user-level network code is doomed to terrible performance.

In this paper we show that it is possible to get adequate performance from a user-level protocol implementation, while retaining all the features of user-level programming that make it far more pleasant than kernel programming.

The key to good performance is the mechanism used to demultiplex received packets to the appropriate user process. Demultiplexing can be done either in the kernel, or in a user-level switching process. User-mode demultiplexing allows flexible control over how packets are distributed, but is expensive because it normally involves at least two context switches and three system calls per received packet. Kernel demultiplexing is efficient, but in existing systems the criteria used to distinguish between packets are too crude.

This paper describes the *packet filter*, a facility that combines both performance and flexibility. The packet filter is part of the operating system kernel, so it delivers packets with a minimum of system calls and context switches, yet it is able to distinguish between packets according to arbitrary and dynamically variable user-specified criteria. The result is a reasonably efficient, easy-to-use abstraction for developing and running network applications.

The facility we describe is not a paper design, but the evolutionary result of much experience and tinkering. The packet filter has been in use at several sites for many years, for both development and production use in a wide variety of applications, and has insulated these applications from substantial changes in the underlying operating system. It has been of clear practical value.

In section 2, we discuss in greater detail the motivation for the packet filter. We describe the abstract interface in section 3, and briefly sketch the implementation in section 4. We then illustrate, in section 5, some uses to which the packet filter has been put, and in section 6 discuss its performance.

2. Motivation

Software to support networking protocols has become tremendously important as a result of use of LAN technology and workstations. The sheer bulk of this software is an indication that it may be overwhelming our ability to create reliable, efficient code: for example, 30% of the 4.3BSD Unix [8, 21] kernel source, 25% of the TOPS-20 [10] (Version 6.1) kernel source, and 32% of the V-system [4] kernel source are devoted to networking.

Development of network software is slow and seldom yields finished systems; debugging may continue long after the software is put into operation. Continual debugging of production code results not only from deficiencies in the original code, but also from inevitable evolution of the protocols and changes in the network environment.

In many operating systems, network code resides in the kernel. This makes it much harder to write and debug:

- Each time a bug is found, the kernel must be recompiled and rebooted.
- Bugs in kernel code are likely to cause system crashes.
- Functionally independent kernel modules may have complex interactions over shared resources.
- Kernel-code debugging cannot be done during normal timesharing; single-user time must be scheduled, resulting in inconvenience for timesharing users and odd work hours for system programmers.
- Sophisticated debugging and monitoring facilities available for developing user-level programs may not be available for developing kernel code.
- Kernel source code is not always available.

In spite of these drawbacks, network code is still usually put in the kernel because the drawbacks of putting it outside the kernel seem worse. If a single user-level process is used for demultiplexing packets, then for each received packet the system will have to switch into the demultiplexing process, notify that process of the packet, then switch again as the demultiplexing process transfers the packet to the appropriate destination process. (Figure 1 depicts the costs associated with this approach.) Context switching and inter-process communication are usually expensive, so clearly it would be more efficient to immediately deliver each packet to the ultimate destination process. (Figure 2 shows how this approach reduces costs.) This requires that the kernel be able to determine to which process each packet should go; the problem is how to allow user-level processes to specify which packets they want.

One simple mechanism is for the kernel to use a specific packet field as a key; a user process registers with the kernel the field value for packets it wants to receive. Since the kernel does not know the structure of higher-layer protocol headers, the discriminant field must be in the lowest layer, such as an Ethernet [9] ‘‘type’’ field. This is not always a good solution. For example, in most environments the Ethernet type field serves only to identify one of a small set of protocol families; almost all packets must be further discriminated by some protocol-specific field. If the kernel can demultiplex only on the type field, then one must still use a user-level switching process with its attendant high cost.

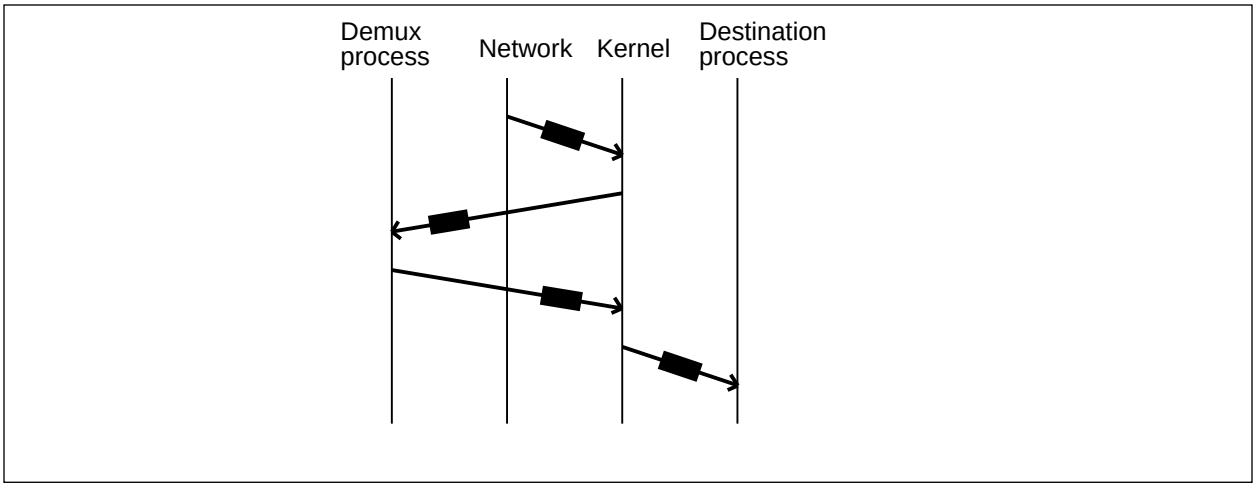


Figure 1: Costs of demultiplexing in a user process

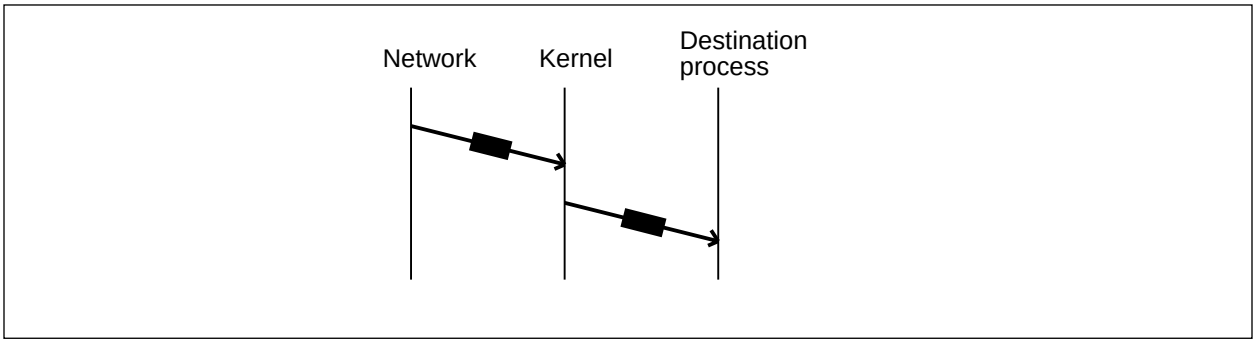


Figure 2: Costs of demultiplexing in the kernel

The packet filter is a more flexible kernel-resident demultiplexer. A user process specifies an arbitrary predicate to select the packets it wants; all protocol-specific knowledge is in the program that receives the packets. There is no need to modify the kernel to support a new protocol. This mechanism evolved for use with Ethernet data-link layers, but will work with most similar datagram networks.

The packet filter not only isolates the kernel from the details of specific protocols; it insulates protocol code from the details of the kernel implementation. The packet filter is not strongly tied to a particular system; in its Unix implementation, it is cleanly separated from other kernel facilities and the novel part of the user-level interface is not specific to Unix. Because protocol code lives outside the kernel it does not have to be modified to be useful with a wide variety of kernel implementations. In systems where context-switching is inexpensive, the performance advantage of kernel demultiplexing will be reduced, but the packet filter may still be a good model for a user-level demultiplexer to emulate.

In addition to the cost and inconvenience of demultiplexing, the cost of domain crossing whenever control crosses between kernel and user domains has discouraged the implementation of protocol code in user processes. In many protocols, far more packets are exchanged at lower levels than are seen at higher levels (these include control, acknowledgement, and duplicate

packets). A kernel-resident implementation confines these overhead packets to the kernel and greatly reduces domain crossing, as depicted in figure 3. The packet filter mechanism cannot eliminate this problem; we can reduce it through careful implementation and by batching together domain-crossing events (see section 3).

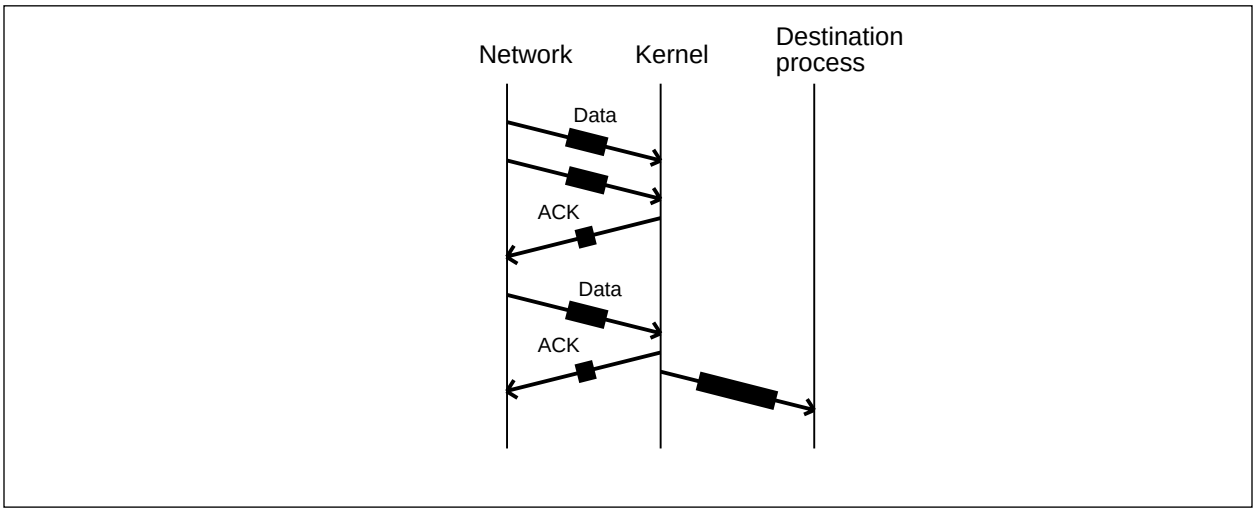


Figure 3: Kernel-resident protocols reduce domain-crossing

User-level access to the data-link layer is not universally regarded as a good thing. Some have suggested that user programs never need access to explicit network communication [23]; others might argue that all networking should be done within a transport protocol such as IP [19] or the ISO Transport Protocol [15], with demultiplexing done by the transport layer code. Both these arguments implicitly assume a homogeneous networking environment, but heterogeneity is often a fact of life: machines from different manufacturers speak various transport protocols, and research on new protocol designs at the data-link level is still profitable.

The packet filter allows rapid development of networking programs, by relatively inexperienced programmers, without disrupting other users of a timesharing system. It places few constraints on the protocols that may be implemented, but in spite of this flexibility it performs well enough for many uses.

2.1. Historical background

As far as we are aware, the idea (and name) of the packet filter first arose in 1976, in the Xerox Alto [3]. Because the Alto operating system shared a single address space with all processes, and because security was not important, the filters were simply procedures in the user-level programs; these procedures were called by the packet demultiplexing mechanism. The first Unix implementation of the packet filter was done in 1980.

3. User-level interface abstraction

Figure 4 shows how the packet filter is related to other parts of a system. Packets received from the network are passed through the packet filter and distributed to user processes; code to implement protocols lives in each process. Figure 5 shows, for contrast, how networking is done

in ‘‘vanilla’’ 4.3BSD Unix; protocols are implemented inside the kernel and data buffers are passed from protocol code to user processes. Figure 6 shows how both models can coexist; some programs may even use both means to access the network.

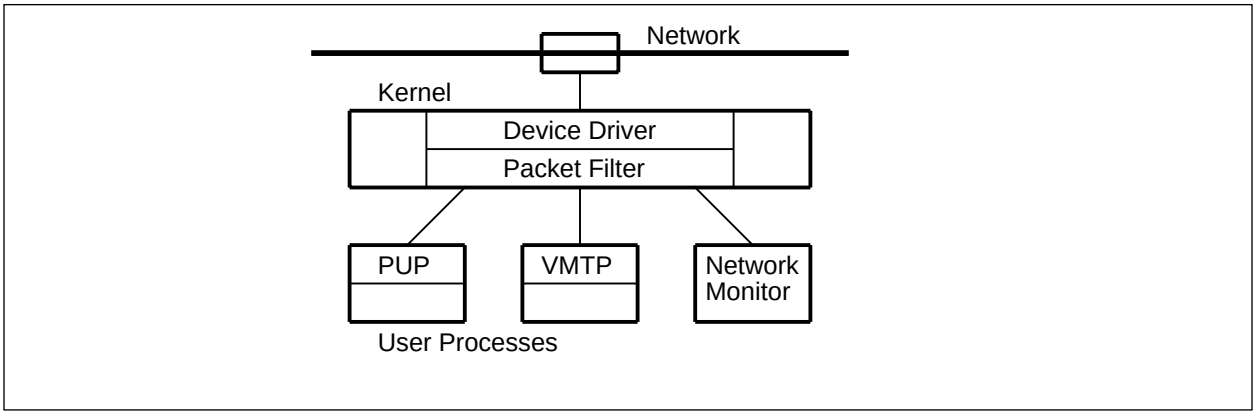


Figure 4: Relationship between packet filter and other system components

The programmer’s interface to the packet filter has three major components: packet transmission, packet reception, and control and status information. We describe these in the context of the 4.3BSD Unix implementation.

Packet transmission is simple; the user presents a buffer containing a complete packet, including data-link header, to the kernel using the normal Unix *write* system call; control returns to the user once the packet is queued for transmission. Transmission is unreliable if the data link is unreliable; the user must discover transmission failure through lack of response rather than an explicit error.

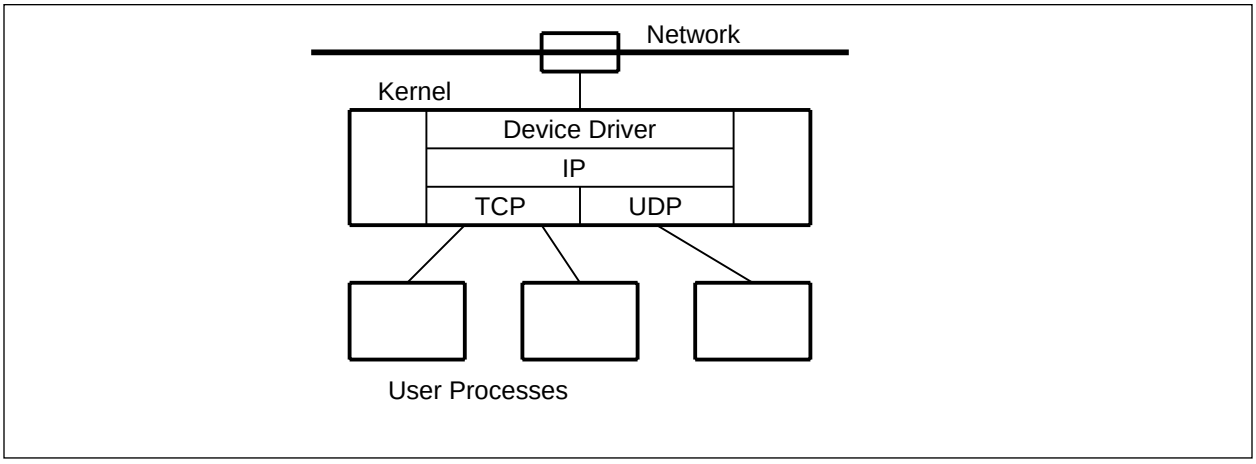


Figure 5: 4.3BSD networking model

Packet reception is more complicated. The packet filter manages some number of *ports*, each of which may be opened by a Unix program as a ‘‘character special device.’’ Associated with each port is a *filter*, a user-specified predicate on received packets. If a filter accepts a packet, the packet is queued for delivery to the associated port. A filter is specified using a small stack-based ‘‘language,’’ in which one can push arbitrary constants or words from the received packet,

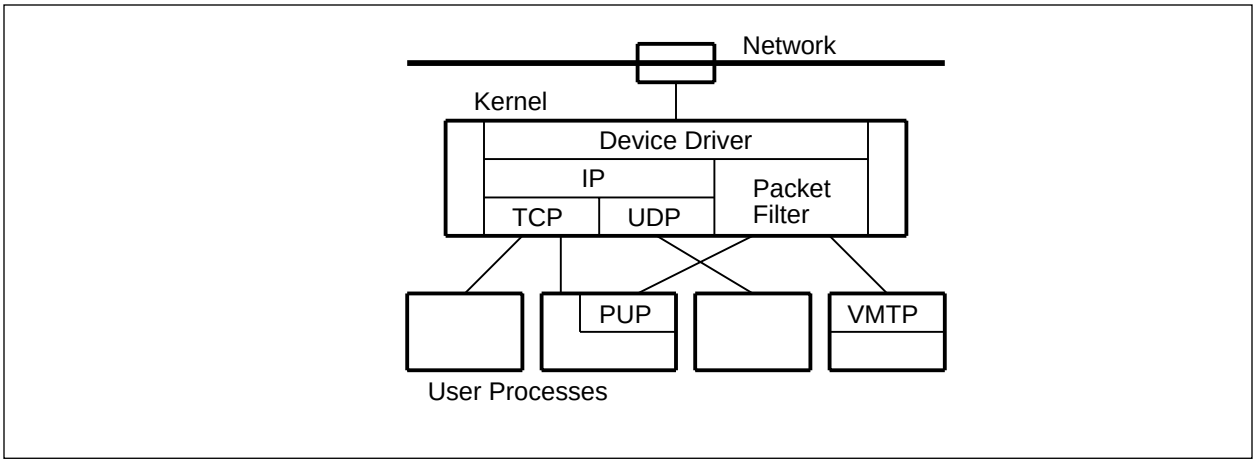


Figure 6: Packet filter coexisting with 4.3BSD networking model

and apply binary operations to the top of the stack. The filter language is discussed in more detail in section 3.1. A process binds a filter to a port using an *ioctl* system call; a new filter can be bound at any time, at a cost comparable to that of receiving a packet; in practice, filters are not replaced very often.

Two processes implementing different communication streams under the same protocol must specify slightly different predicates so that packets are delivered appropriately. For example, a program implementing a Pup [2] protocol would include a test on the Pup destination socket number as part of its predicate. The layering in a protocol architecture is not necessarily reflected in a filter predicate, which may well examine packet fields from several layers.

When a program performs a *read* system call on the file descriptor corresponding to a packet filter port, the first of any queued packets is returned. The entire packet, including the data-link layer header, is returned, so that user programs may implement protocols that depend on header information. The program may ask that all pending packets be returned in a batch; this is useful for high-volume communications because it can amortize the overhead of performing a system call over several packets. Figure 7 depicts per-packet overheads without batching; figure 8 shows how these are reduced when batching is used.

If no packets are queued, the *read* system call blocks until a packet is available; if no packet arrives during a timeout period, the *read* call terminates and reports an error. Simple programs can be written using a ‘‘write; read with timeout; retry if necessary’’ paradigm. More elaborate programs may take advantage of two more sophisticated synchronization mechanisms: the 4.3BSD *select* system call, or a interrupt-like facility using Unix ‘‘signals,’’ either of which allows non-blocking network I/O.

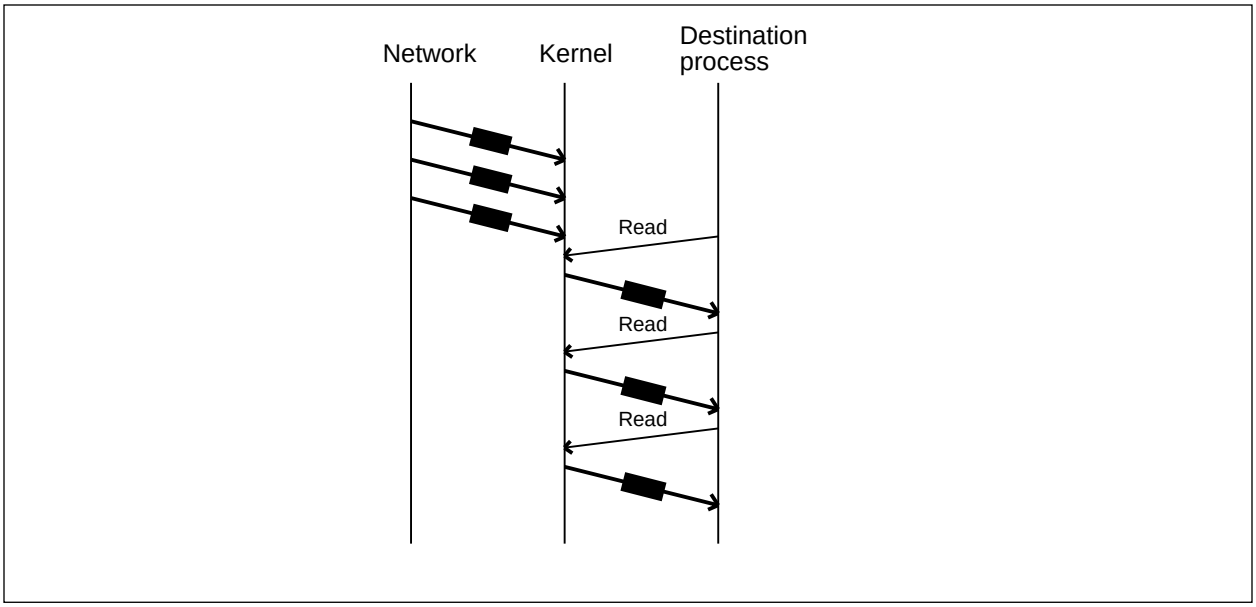


Figure 7: Delivery without received-packet batching

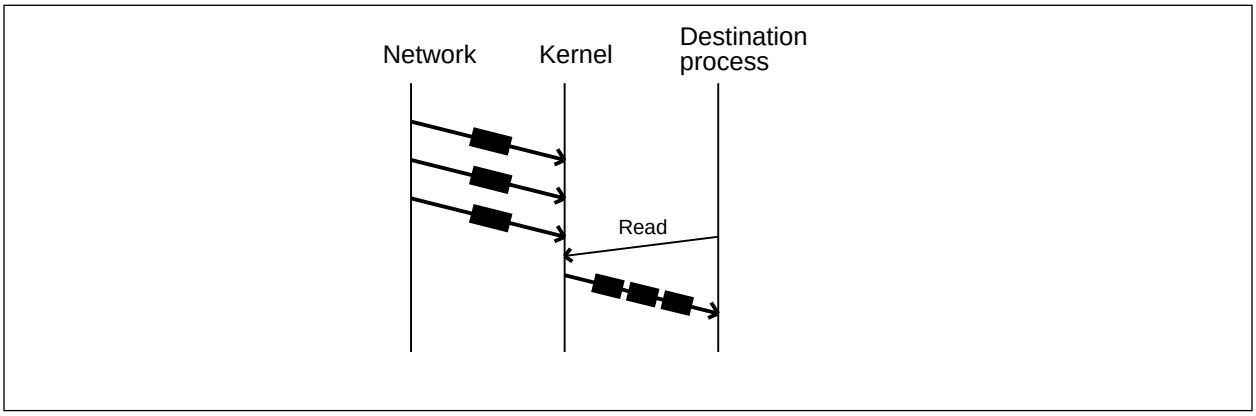


Figure 8: Delivery with received-packet batching

3.1. Filter language details

The heart of the packet filter is an interpreter for the ‘‘language’’ shown in figure 9. A filter is a data structure including an array of 16-bit words. Each word is normally interpreted as an instruction with two fields, a *stack action* field and a *binary operation* field.

A stack action may cause either a word from the received packet or a constant to be pushed on the stack. A binary operation pops the top two words from the stack, and pushes a result. Thus, filter programs evaluate a logical expression composed of tests on the values of various fields in the received packet. The filter is normally evaluated until the program is exhausted. If the value remaining on top of the stack is non-zero, the filter is deemed to have accepted the packet.

It is sometimes possible to avoid evaluating the entire filter before deciding whether to accept a packet. This is especially important for performance, since on a busy system several dozen filters may be applied to an incoming packet before it is accepted. The filter language therefore includes four ‘‘short-circuit’’ binary logical operations, that when evaluated either push a result and allow the program to continue, or terminate the program and return an appropriate boolean.

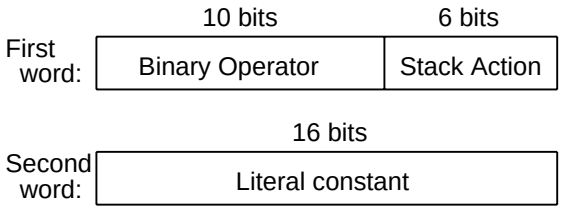
Figure 11 shows an example of a simple filter program; figure 12 shows an example of a filter program using short-circuit operations. Both are used with Pup [2] packets on a 3Mbit/sec. Experimental Ethernet [17]; the data-link header is 4 bytes (two words) long, with the packet type in the second word (see figure 10.) In normal use, the filters are not directly constructed by the programmer, but are ‘‘compiled’’ at run time by a library procedure.

The design of the filter language is not the result of careful analysis but rather embodies several accidents of history, such as its bias towards 16-bit fields. It has evolved over the years; in particular, the short-circuit operations were added after an analysis showed that they would reduce the cost of interpreting filter predicates. One could imagine alternatives to the stack language structure; for example, a predicate could be an array of (*field-offset*, *expected-value*) pairs, and the predicate would be satisfied if all the specified fields had the specified values. However, the additional flexibility of the stack language has often proved useful in constructing efficient filters. Since the ‘‘instruction set’’ is implemented in software, not hardware, there is no execution-time penalty associated with supporting a broad range of operations.

3.2. Access Control

Normally, once a packet has been accepted for delivery to a process, it will not be submitted to the filters of any other processes. Because it is not possible to determine if two filters will accept overlapping sets of packets, we need a way to prevent one process from inappropriately diverting packets meant for another process.

Associated with each filter is a *priority*; filters are applied in order of decreasing priority, so if two filters would both accept a packet, it goes to the one with higher priority. (Priority has another purpose; if priorities are assigned proportional to the likelihood that a filter will accept a packet, then the ‘‘average’’ packet will match one of the first few filters it is tested against, consequently reducing the amount of filter interpretation overhead.) If two filters have the same priority, the order of application is unspecified (the interpreter may occasionally reorder such filters to place the busier ones first); in these cases one must take care to ensure that the filters accept disjoint sets of packets.



(second word used only if Stack Action = PUSHLIT)

Instruction format

Stack Action	Effect on stack
NOPUSH	None
PUSHLIT	Following instruction word is pushed
PUSHZERO	Constant zero is pushed
PUSHONE	Constant one is pushed
PUSHFFFF	Constant 0xFFFF is pushed
PUSHFF00	Constant 0xFF00 is pushed
PUSH00FF	Constant 0x00FF is pushed
PUSHWORD+*n*	*n*th word of packet is pushed

All binary operations except NOP remove two words from the top of the stack and push one result word. In the table that follows, the original top of stack is abbreviated T1, the word below that is T2, and the result is R. For logical operations (AND, OR, XOR), a value is interpreted as TRUE if it is non-zero.

Binary Operation	Result on stack
EQ	R := TRUE if T2 == T1, else FALSE
NEQ	R := TRUE if T2 <> T1, else FALSE
LT	R := TRUE if T2 < T1, else FALSE
LE	R := TRUE if T2 <= T1, else FALSE
GT	R := TRUE if T2 > T1, else FALSE
GE	R := TRUE if T2 >= T1, else FALSE
AND	R := T2 AND T1
OR	R := T2 OR T1
XOR	R := T2 XOR T1
NOP	No effect on stack

The following ‘‘short-circuit’’ binary operations all evaluate R := (T1 == T2) and push the result R on the stack. They return immediately under specified conditions, otherwise the program continues.

Binary operation	Returns immediately	if result is
COR	TRUE	TRUE
CAND	FALSE	FALSE
CNOR	FALSE	TRUE
CNAND	TRUE	FALSE

Figure 9: Packet filter language summary

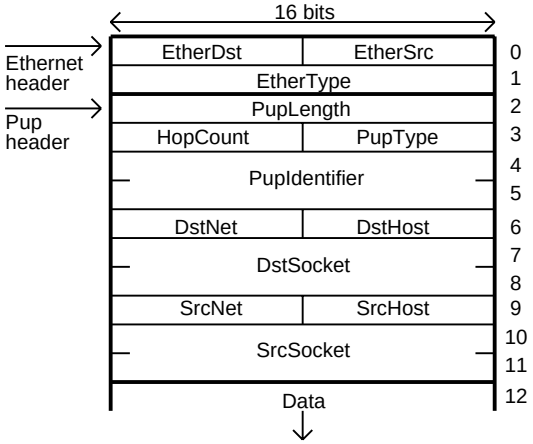


Figure 10: Format of Pup Packet header on 3Mb Ethernet (after [2])

This filter accepts all Pup packets with Pup Types between 1 and 100. The Pup Type field is a one byte field, so it must be masked out of the appropriate word in the packet.

```
struct enfilter f = {
    10, 12,                          /* priority and length */
    PUSHWORD+1, PUSHLIT | EQ, 2,     /* packet type == PUP */
    PUSHWORD+3, PUSH00FF | AND,      /* mask low byte */
    PUSHZERO | GT,                   /* PupType > 0 */
    PUSHWORD+3, PUSH00FF | AND,      /* mask low byte */
    PUSHLIT | LE, 100,               /* PupType <= 100 */
    AND,                             /* 0 < PupType <= 100 */
    AND                              /* && packet type == PUP */
};
```

Figure 11: Example filter program

This filter accepts Pup packets with a Pup DstSocket field of 35. The DstSocket field occupies two words, so the filter must test both words and combine them with an AND operation. The DstSocket field is checked before the packet type field, since in most packets the DstSocket is likely not to match and so the short-circuit operation will exit immediately.

```
struct enfilter f = {
    10, 8,                           /* priority and length */
    PUSHWORD+8, PUSHLIT | CAND, 35,  /* Low word of socket == 35 */
    PUSHWORD+7, PUSHZERO | CAND,     /* High word of socket == 0 */
    PUSHWORD+1, PUSHLIT | EQ, 2      /* packet type == Pup */
};
```

Figure 12: Example filter program using short-circuit operations

Optionally, a process may specify that the packets accepted by its filter should be submitted to other, lower-priority, filters as well; multiple copies of such packets may be delivered. This is useful in implementing monitoring facilities without disturbing the processes being monitored, in ‘‘group’’ communication where a packet may be multicast to several processes on one host, or when it is not possible to filter precisely enough within the kernel.

This access control mechanism does not in itself protect against malicious or erroneous processes attempting to divert packets; it only works when processes play by the rules. In the research environment for which the packet filter was developed, this has not been a problem, especially since there are many other ways to eavesdrop on an Ethernet. An earlier version of the packet filter did provide some security by restricting the use of high-priority filters to certain users, allowing these users first rights to all packets, but this mechanism went unused.

Because typical networks are easily tapped, most proposals for secure communication rely on encryption to protect against eavesdropping. If packets are encrypted, some header fields must be transmitted in cleartext to allow demultiplexing; this is not peculiar to use of the packet filter, especially if encryption is on a per-process basis.

3.3. Control and status information

The user can control the packet filter’s action in a variety of ways, by specifying: the filter to be associated with a packet filter port; the timeout duration for blocking reads (or optionally, immediate return or indefinite blocking); the signal, if any, to be delivered upon packet reception; and the maximum length of the per-port input queue.

Information provided by the packet filter to programs includes: the type of the underlying data-link layer; the lengths of a data-link layer address and of a data-link layer header; the maximum packet size for the data-link; the data-link address for incoming packets; and the address used for data-link layer broadcasts, if one exists.

The user can also ask that each received packet be marked with a timestamp and a count of the number of packets lost due to queue overflows in the network interface and in the kernel.

4. Implementation

The packet filter is implemented in 4.3BSD Unix as a ‘‘character special device’’ driver. Just as the Unix terminal driver is layered above communications device drivers to provide a uniform abstraction, the packet filter is layered above network interface device drivers. As with any character device driver, it is called from user code via *open*, *close*, *read*, *write*, and *ioctl* system calls. The packet filter is called from the network interface drivers upon receipt of packets not destined for kernel-resident protocols.

Most of the complexity in the implementation is involved in bookkeeping and in managing asynchrony. When a packet is received, it is checked against each filter, in order of decreasing priority, until it is accepted or until all filters have rejected it (see figure 13). The filter interpreter is straightforward, but must be carefully coded since its inner loop is quite busy. It simply iterates through the ‘‘instruction words’’ of a filter (there are no branch instructions), evaluating the filter predicate using a small stack. When it reaches the end of the filter, or a short-circuit conditional is satisfied, or an error is detected, it returns the predicate value to indicate acceptance or rejection of the packet.

```
Accepted := false;
for priority := MaxPriority downto MinPriority do
  for i := FirstFilter[priority] to
                LastFilter[priority] do
    if Apply(Filter[i], rcvd_pkt) = MATCH then
      Deliver(Port[i], rcvd_pkt);
      Accepted := true;
    end;
  end;
end;
if not Accepted then
    Drop(rcvd_pkt);
end;
```

Figure 13: Pseudo-code for filter application loop

The packet filter module is about 2000 lines of heavily-commented C source code (under 6K bytes of Vax machine code); each of the network interface device drivers must be modified with a few dozen lines of linkage code. Aside from this, the packet filter requires no modification of the Unix kernel. Because it is well-isolated from the rest of the kernel, it is easily ported to different Unix implementations. Ports have been made to the Sun Microsystems Inc. operating system, which is internally quite similar to 4.2BSD, and to the Ridge Operating System (ROS) of Ridge Computers, Inc. ROS is a message-based operating system with inexpensive processes [1]; its internal structure is distinctly different from that of Unix. The packet filter has also been ported to Pyramid Technology’s Unix system, with minor modification for use in a multi-processor. It appears to be relatively easy to port the packet filter to a variety of operating systems; this in turn makes it possible to port user-level networking code without further kernel modifications.

5. Uses of the packet filter

The packet filter is successful because it provides a useful facility with adequate performance. Section 6 provides quantitative measures of performance; in this section we consider qualitative utility.

The primary goal of the packet filter is to simplify the development and improvement of networking software and protocols. Since networking software is often in a continual state of development, anything that speeds debugging and modification reduces the mismatch between the software and the networking environment. This is especially important for the experimental development of new protocols. Similarly, since operating systems are continually changing, decoupling network code from the rest of the system reduces the risk of ‘‘software rot.’’

The remainder of this section describes examples demonstrating how the packet filter has been of practical use.

5.1. Pup protocols

The Pup [2] protocol suite includes a variety of applications using both datagram (request-response) and stream transport protocols. At Stanford, almost all of the Pup protocols were implemented for Unix, based entirely on the packet filter. Although Pup, as an experimental architecture, has some notable flaws, for about five years this implementation served as the primary link between Stanford’s Unix systems and other campus hosts and workstations. Pup is still in relatively heavy use in a number of organizations, most of which have used the Stanford implementation.

The experience with Pup has shown the value of decoupling the networking implementation from the Unix kernel. Not only did this make it possible to develop the Pup code without the effort of kernel debugging, it also made it possible to modify the kernel without having to worry about the integrity of the Pup code. When, every few years, a new release of the Berkeley Unix kernel became available, it sufficed to re-install the kernel module implementing the packet filter. The Pup code could then be run, often without recompilation, under the new operating system. The initial port of the packet filter code from 4.1BSD to 4.2BSD took several evenings; for comparison, it took six programmer-months to port BBN’s TCP implementation from 4.1BSD to 4.2BSD [14]. That the BBN TCP code is kernel-resident undoubtedly contributed to the time it took to port.

5.2. V-system protocols

The V-system is a message-based distributed operating system. As an ongoing research project, it is under continual development and revision. The architects of the V-system have chosen to design their own protocols, to obtain high performance and so that they could make use of the multicast feature of Ethernet hardware [6].

Although the V-system is primarily a collection of workstations and servers running the V kernel, Unix hosts were integrated into the distributed system to provide disk storage, compute cycles, mail service, and other amenities not available in a new operating system. The Unix hosts had to be taught to speak the V-system Inter-Kernel Protocol (IKP). Fortunately, the packet filter was available for use as the basis of a user-level V IKP server process.

The V IKP is a simple protocol and could have been put in the Unix kernel. This, however, would have required the V researchers to learn about the details of the Unix kernel, to participate in the maintenance of the kernel, and to re-install the IKP implementation in each new release of the operating system. Instead, they were able to devote their attention to research on the topics that interested them. One result of this research was the VMTP protocol [5], a replacement for the V IKP. Although there is a kernel-resident implementation of VMTP for 4.3BSD, the first implementation used the packet filter. The user-level implementation allowed rapid development of the protocol specification through experimentation with easily-modified code. (Section 6.3 contrasts the performance differences between the two VMTP implementations.)

5.3. RARP

The Reverse Address Resolution Protocol (RARP) [12] was designed to allow workstations to determine their Internet Protocol (IP) addresses without relying on any local stable storage. One issue in the definition of this protocol was whether it should be a layer above IP, or a parallel layer. The former leads to a chicken-or-egg dilemma; the latter is cleaner but raised question of implementability under 4.2BSD. With the packet filter, however, a RARP implementation was easy; the work was done in a few weeks by a student who had no experience with network programming, and who had no need to learn how to modify the Unix kernel.

5.4. Network Monitoring

For the developer or maintainer of network software, no tool is as valuable as a *network monitor*. A network monitor captures and displays traces of the packets flowing among hosts; a packet trace makes it much easier to understand why two hosts are unable to communicate, or why performance is not up to par.

Most commercially-available network monitors (including the Excelan *LANalyzer* [11], the Network General *Sniffer* [18], and the Communications Machinery Corp. *LanScan* [7]) are stand-alone units dedicated to monitoring specific protocols. A network monitor closely integrated with a general-purpose operating system, running on a workstation, has several important advantages over a dedicated monitor:

- All the tools of the workstation are available for manipulating and analyzing packet traces.
- A user can write new monitoring programs to display data in novel ways, or to monitor new or unusual protocols.

One of us has been using the packet filter, on a MicroVAX-II workstation, as the basis for a variety of experimental network monitoring tools. This system has sufficient performance to record all packets flowing on a moderately busy Ethernet (with rare lapses), and more than sufficient performance to capture all packets between a pair of communicating hosts. Since one can easily write arbitrarily elaborate programs to analyze the trace data, and even to do substantial analysis in real time, an integrated network monitor appears to be far more useful than a dedicated one. (Sun Microsystems' *etherfind* program is another example of an integrated network monitor. It is based on Sun's *Network Interface Tap* (NIT) facility, which is similar to the packet filter but only allows filtering on a single packet field[1] [22].)

[1]Sun expects to include our packet-filtering mechanism in a future release of NIT.

6. Performance

We measured the performance of the packet filter in several ways. We determined the amount of processor time spent on packet filter routines, and we measured the throughput of protocol implementations based on the packet filter. We compared these measurements with those for kernel-resident implementations of similar protocols, and found that in practice packet-filter-based protocol implementations perform fairly well.

All measurements were made using VAX processors running 4.2BSD or 4.3BSD Unix, using either a 10Mbit/sec or 3Mbit/sec Ethernet. Note that the packet filter coexists with kernel-resident protocol implementations, without affecting their performance.

6.1. Kernel per-packet processing time

One indication of the packet filter's cost is the kernel CPU time required to process an ''average'' received packet. We measured this time for the packet filter, and for analogous functions of kernel-resident protocols. A 4.3BSD Unix kernel was configured to collect the CPU time spent in and number of calls made to each kernel subroutine. The profiled kernel was run for 28 hours on a timesharing VAX-11/780, and *gprof* [13] was used to format the data.

During the profiling period, the system handled 1.3 million packets. 21% of these packets were processed by the packet filter; of the remainder, 69% were IP packets and 10% were ARP packets. All per-packet processing times reported are for ''average'' packets and ''typical'' filter predicates.

Processing times for transmitted packets are about the same for either the packet filter or the kernel-resident IP implementation; it takes about 1 mSec to send a datagram. The packet filter has a slight edge, since it does not need to choose a route for the datagram or compute a checksum.

Packet filter: The packet filter spends an average of 1.57 mSec processing each packet. 41% of this time is spent evaluating filter predicates; the average packet is tested against 6.3 predicates. We derived a crude estimate for the time to process a packet: 0.8 mSec + (0.122 * *number of predicates tested*) mSec. The average number of predicates tested will normally be somewhat less than half the number of active ports, because the priority mechanism described in section 3.2 can cause the most likely filters to be tested first.

Kernel-resident IP implementation: The average time required to process a received IP packet was 1.77 mSec. This includes all protocol processing up to the TCP and UDP layers; if only the IP layer processing is counted, the average packet requires about 0.49 mSec. This means that the kernel-resident IP layer is about three times faster than the packet filter at processing an average packet.

6.2. Total per-packet processing time

The kernel profile does not account for the entire cost of handling packets. We measured actual packet rates into and out of user processes on a microVax-II running Ultrix 1.2, using a synthetic load. The results for packet reception are included in tables 8 and 9 in section 6.5.

Although sending datagrams via the packet filter costs less than sending an unchecksummed UDP datagram of the same size (see table 1), we estimate that this is still about twice the cost for

the kernel to send a datagram on its own. For packets that carry no useful data (acknowledgements, for example) user-level protocol implementations pay this additional penalty.

Total packet size	Elapsed time per packet sent via packet filter	Elapsed time per packet sent via UDP
128 bytes	1.9 mSec	3.1 mSec
1500 bytes	3.6 mSec	4.9 mSec

Table 1: Cost of sending packets

6.3. VMTP performance

The only interesting protocol for which there is both a packet-filter based implementation and a kernel-resident implementation is VMTP [5]. This provides a basis for a direct measurement of the cost of user-level implementation; while there are minor differences in the actual protocols implemented, and the two implementations are not of precisely equal quality, they follow essentially the same pattern of packet transport. All these measurements, unless noted, were carried out using microVax-II processors, 4.3BSD Unix, and a 10Mbit/sec Ethernet. In each case, both ends of the transfer used identical protocol implementations.

We measured the cost for a minimal round-trip operation (reading zero bytes from a file). The results, shown in table 2 (and figure 14), indicate that the penalty for user-level implementation is almost exactly a factor of two. On this measurement, the Unix kernel implementation of VMTP is quite close to the V kernel implementation, indicating that there is no obvious inefficiency in the Unix kernel implementation.

VMTP Implementation	Elapsed time per operation
Packet filter	14.7 mSec
Unix kernel	7.44 mSec
V kernel	7.32 mSec

Table 2: Relative performance of VMTP for small messages

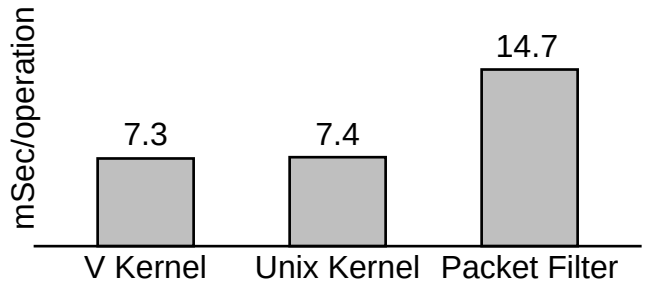


Figure 14: Relative performance of VMTP for small messages

We also measured the cost for transferring bulk data using VMTP. This was done by repeatedly reading the same segment of a file, which therefore stayed in the file system buffer

cache; consequently, the measured rates should be nearly independent of disk I/O speed. (In each trial about 1 Mb was transferred.) We also measured TCP performance, for comparison; note that TCP checksums all data, whereas these implementations of VTMP do not. The results, shown in table 3 (and in figure 15), show that in this case the penalty for user-level implementation of VMTP is almost exactly a factor of three.

Implementation	Rate
Packet filter	112 Kbytes/sec
Unix kernel VMTP	336 Kbytes/sec
V kernel VMTP	278 Kbytes/sec
Unix kernel TCP	222 Kbytes/sec

Table 3: Relative performance of VMTP for bulk data transfer

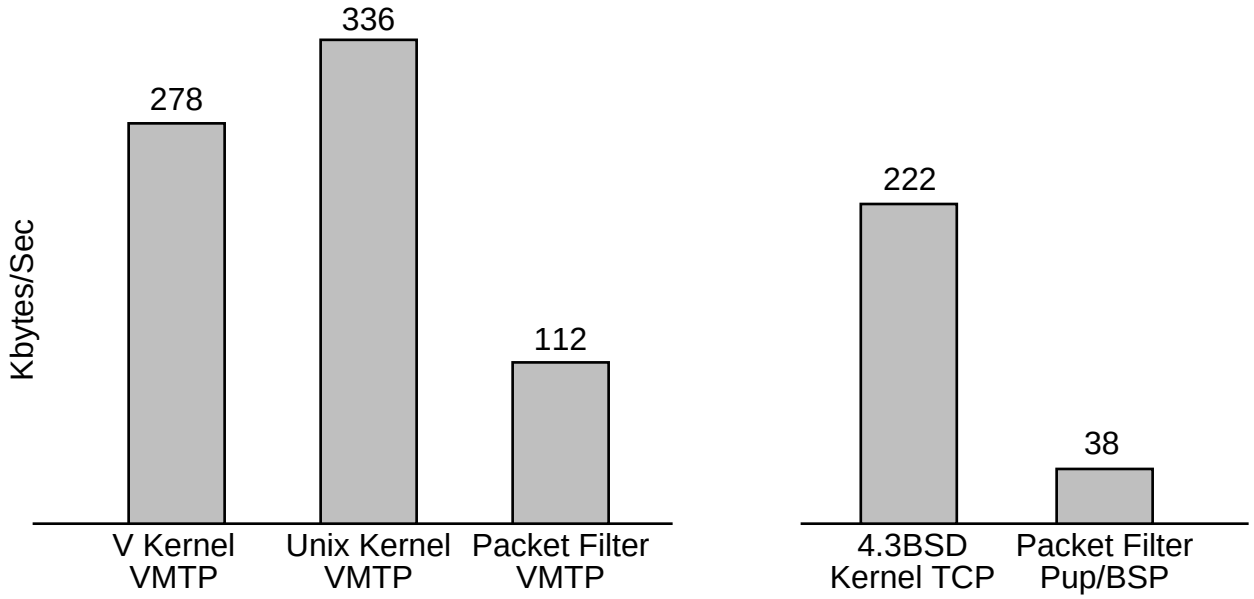


Figure 15: Relative performance of VMTP for bulk data transfer

The packet-filter based implementation measured in table 3 uses received-packet batching. Table 4 (and figure 16) shows that batching improves throughput by about 75% over identical code that reads just one packet per system call; the difference cannot be entirely due to decreased system call overhead, but may reflect reductions in context switching and dropped packets.

Batching	Rate
Yes	112 Kbytes/sec
No	64 Kbytes/sec

Table 4: Effect of received-packet batching on performance

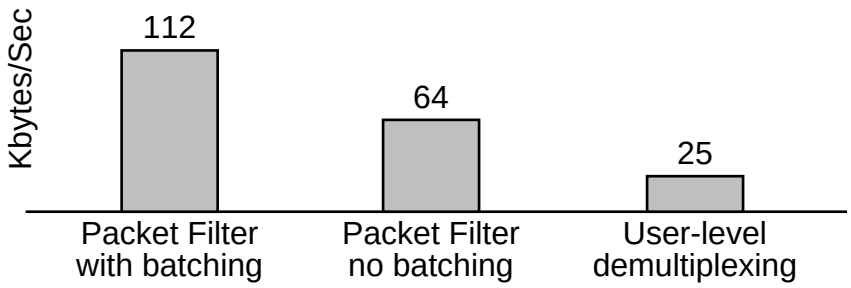


Figure 16: Effect of received-packet batching and user-level demultiplexing on performance

We also tried to measure the cost of a user-level demultiplexing process, by simulating it within the client VTMP implementation. This is done by using an extra process to receive packets, which are then passed to the actual VMTP process via a Unix pipe. (In this case, the server process was *not* modified.) Table 5 (and figures 16 and 17) shows that user-level demultiplexing has a small cost (20% greater latency) for short messages, but decreases bulk throughput by more than a factor of four (much of this is attributable to the poor IPC facilities in 4.3BSD).

Demultiplexing done in	Elapsed time per minimal operation	Bulk rate
Kernel	14.72	112 Kbytes/sec
User process	18.08	25 Kbytes/sec

Table 5: Effect of user-level demultiplexing on performance

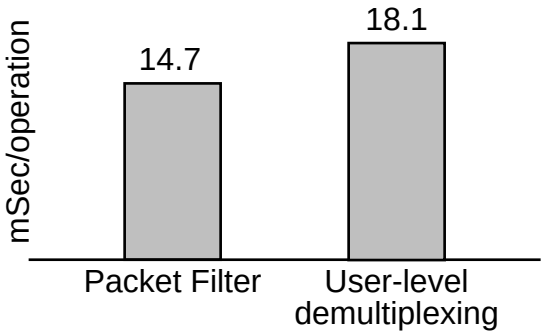


Figure 17: Effect of user-level demultiplexing for small VMTP messages

6.4. Byte-stream throughput

We compared the performance of a Pup/BSP (Byte-Stream Protocol) implementation using the packet filter with that of the IP/TCP [20] implementation in the 4.3BSD kernel. These measurements were carried out using microVax-II processors, 4.3BSD Unix, and a 10Mbit/sec Ethernet.

Table 6 (and figure 15) shows the rates at which the two implementations can transfer bulk data from process to process. TCP is faster by almost a factor of six. When used to implement a File Transfer Protocol (FTP), TCP slows by a factor of two if the source of data is a disk file, but the BSP throughput remains unchanged, indicating that network performance is the rate-limiting factor for BSP file transfer.

Implementation	Rate
Packet filter BSP	38 Kbytes/sec
Unix kernel TCP	222 Kbytes/sec

Table 6: Relative performance of stream protocol implementations

Pup (hence BSP) allows a maximum packet size of 568 bytes, whereas TCP in 4.3BSD uses 1078-byte packets and so sends only half as many; we found that if TCP is forced to use the smaller packet size, its performance is cut in half. After this correction, TCP throughput is still three times that of BSP; most of difference is attributable to the cost of BSP's user-level implementation. This is consistent with the factor-of-two difference we measured for VMTP.

We also measured performance for Telnet (remote terminal access)[2]. A program on the ''server'' host (Vax-11/780) prints characters which are transmitted across the network and displayed at the ''user'' host. The results are shown in table 7. The ''Output rate'' column shows the overall throughput, in characters per second, for each configuration.

Telnet protocol	Network bandwidth	Output rate
Pup/BSP	10 Mbit/sec	1635
IP/TCP	10 Mbit/sec	1757
Pup/BSP	3 Mbit/sec	878
IP/TCP	3 Mbit/sec	933

Table 7: Relative performance of Telnet

The first two rows of the table show throughput using an MC68010-based workstation capable of displaying about 3350 characters per second. The achieved throughput is about half that, varying only slightly according to whether TCP or BSP (and thus the packet filter) is used. The last two rows, measured with characters displayed on a 9600 baud terminal, show almost no difference between BSP and TCP performance. These output rates are clearly limited by the display terminal, not by network performance.

In summary, a kernel-resident implementation of a stream protocol such as VMTP or BSP appears to be about two or three times as fast as an implementation based on the packet filter. In many applications, the actual performance difference may be much smaller; the packet-filter implementation of VMTP is only 40% slower than the kernel-resident TCP when used for file transfer. The VMTP and BSP implementations are quite useful in practice; disks and terminals are more often serious bottlenecks than the packet filter.

[2]This test was done under 4.2BSD.

6.5. Costs of demultiplexing outside the kernel

We asserted in section 2 that using a user-level process to demultiplex received packets to other processes would result in poor performance. In section 6.3 we showed that this appears to be true, especially for bulk-data transfer. In this section, we analyze the additional cost using measurements of Ultrix 1.2; the measurements are inspired by those made by McKusick, Karels, and Leffler [16].

6.5.1. Analytical model

If a demultiplexing process is used, each received packet results in at least two context switches: one into the demultiplexing process and one into the receiving process[3]. If the system has other active processes, an additional context switch to an unrelated process may occur, when the receiving process blocks waiting for the next packet.

With direct delivery of received packets, in the best case the receiving process will never be suspended, and no context switches take place. In the worst case, with other active processes, a received packet will cause two context switches.

Either mechanism requires at least one data transfer between kernel and process. Since Unix does not support memory sharing, the demultiplexing process requires two additional data transfers to get the packet into the final receiving process.

6.5.2. Cost of overhead operations

Benchmarks indicate that a MicroVAX-II running Ultrix 1.2 requires about 0.4 mSec of CPU time to switch between processes, and about 0.5 mSec of CPU time to transfer a short packet between the kernel and a process. Therefore, we predict that receiving a short packet using a demultiplexing process should take at least 2.3 mSec while for the packet filter, these overhead costs may be as low as 0.5 mSec per packet; the difference increases for longer packets because data copying requires about 1 mSec/Kbyte.

6.5.3. Measured costs

These costs are not the only ones associated with receiving a packet; they are the ones that are affected by the use of user-level demultiplexing. We measured the actual elapsed time required to receive packets of various sizes; the ‘‘demultiplexing process’’ receives packets from the network and passes them to a second process via a Unix pipe. The results are shown in table 8. The additional cost of user-level demultiplexing agrees fairly closely with our predication.

Packet size	Elapsed time if demultiplexing done in kernel	Elapsed time if demultiplexing done in user process
128 bytes	2.3 mSec	5.0 mSec
1500 bytes	4.0 mSec	9.0 mSec

Table 8: Per-packet cost of user-level demultiplexing

[3]We assume that no batching of packets takes place; this assumption breaks down when packets arrive faster than the system can switch contexts.

Since received-packet batching, as we saw in section 6.3, can amortize the costs of context-switching over many packets, we repeated our measurements with batching enabled; the batch size was hard to control but the results are about the same for four or more packets per batch. The results are shown in table 9; batching clearly reduces the penalty associated with user-level demultiplexing, but the difference remains significant.

Packet size	Elapsed time if demultiplexing done in kernel	Elapsed time if demultiplexing done in user process
128 bytes	1.9 mSec	2.4 mSec
1500 bytes	3.5 mSec	5.9 mSec

Table 9: Per-packet cost of user-level demultiplexing with received-packet batching

The measurements in tables 8 and 9 were made without any real decision-making on the part of the demultiplexer. Before we condemn user-level demultiplexing on the basis of its high overhead, we must show that the cost of interpreting packet filters in the kernel does not dwarf the benefit of avoiding context switches (presumably, a user-level demultiplexer could make decisions at least as efficiently and possibly more so). We measured the cost of interpreting filter programs of various lengths; the results are shown in table 10. (Batching was enabled and all packets were 128 bytes long.) It usually takes two or three filter instructions to test one packet field; even with rather long filters (21 instructions) the additional cost for filter interpretation is less than the cost of user-level demultiplexing if no more than three such long filters are applied to an incoming packet before one filter accepts it.

Filter length (instructions)	Elapsed time per packet
0	1.9 mSec
1	2.0 mSec
9	2.2 mSec
21	2.5 mSec

Table 10: Cost of interpreting packet filters

For filters using short-circuit conditionals, the break-even point is closer to an average of about ten filters before acceptance, which should occur when more than twenty filters are active. This means that even if one assumes zero cost for decision-making in a user-level demultiplexer, the break-even point comes with twenty different processes using the network. For packets longer than 128 bytes, the break-even point comes with even more active processes.

In summary, kernel demultiplexing performs significantly better than user-level demultiplexing for a wide range of situations. This advantage disappears only if a very large number of processes are receiving packets.

7. Problems and possible improvements

Since its beginnings in early 1980, the packet filter has often been revised to support additional applications or provide better performance. There is still room for improvement.

The filter language described in section 3 only allows the user to specify packet fields at constant offsets from the beginning of a packet. This has been adequate for protocols with fixed-format headers (such as Pup), but many network protocols allow variable-format headers. For example, since the IP header may include optional fields, fields in higher layer protocol headers are not at constant offsets. The current packet filter can be made to handle non-constant offsets only with considerable awkwardness and inefficiency; the filter language needs to be extended to include an ‘‘indirect push’’ operator, as well as arithmetic operators to assist in addressing-unit conversions.

The current filter mechanism deals with 16-bit values, requiring multiple filter instructions to load packet fields that are wider or narrower. It is possible that direct support for other field sizes would improve filter-evaluation efficiency. The existing read-batching mechanism clearly improves performance for bulk data transfer; a write-batching option (to send several packets in one system call) might also improve performance.

In addition to these problems, which may be regarded as deficiencies in the abstract interface, there is room for improvement in the existing implementation. During evaluation of each filter instruction, the interpreter verifies that the instruction is valid, that it doesn’t overflow or underflow the evaluation stack, and that it doesn’t refer to a field outside the current packet. Since the filter language does not include branching instructions, all these tests can be performed ahead of time (except for indirect-push instructions); this might significantly speed filter evaluation. Even more speed could be gained by compiling filters into machine code, at the cost of greatly increased implementation complexity. Finally, with a redesigned filter language it might be possible to compile the set of active filters into a decision table, which should provide the best possible performance.

Idiosyncrasies of the 4.3BSD kernel create other inefficiencies. For example, because 4.3BSD network interface drivers strip the data-link layer header from incoming packets, the packet filter may be spending a significant amount of time to restore these headers. Also, in order to mark each packet with a unique timestamp, the packet filter calls a kernel subroutine called *microtime*; on a VAX-11/780, this costs about 70 uSec, probably more than the timestamp is worth.

8. Summary

The performance of the packet filter is clearly better than that of a user-level demultiplexer, and the performance of protocol code based on the packet filter is clearly worse than that of kernel-resident protocol code. Since the packet filter is just as flexible as a user-level demultiplexer, we believe that in systems where context-switching has a substantial cost, it is the right basis for implementing network code outside the kernel.

Are the advantages of user-level network code, even with the packet filter, worth the extra cost? Our experience has convinced us that in many cases, it is. The performance of such code is quite acceptable, and it greatly eases the task of developing protocols and their implementations. The packet filter appears to put just enough mechanism in the kernel to provide decent performance, while retaining the flexibility of a user-level demultiplexer.

Acknowledgments

Many people have used or worked on the packet filter implementation over the years; without their support and comments it would not be nearly as useful as it is. Especially notable are those who ported the code to other operating systems: Jon Reichbach of Ridge Computers, Inc., Glenn Skinner of Sun Microsystems, Inc., and Charles Hedrick of Rutgers University, who ported it to Pyramid Technology's system. Steve Deering and Ross Finlayson of Stanford made the VMTP measurements possible. We would like to thank the program committee and student reviewers for their comments.

References

[1] Ed Basart.
The Ridge Operating System: High performance through message-passing and virtual memory.
In *Proceedings of the 1st International Conference on Computer Workstations*, pages 134-143. IEEE, November, 1985.

[2] David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe.
Pup: An internetwork architecture.
IEEE Transactions on Communications COM-28(4):612-624, April, 1980.

[3] David Boggs and Edward Taft.
Private communication.
1987.

[4] David R. Cheriton.
The V Kernel: A software base for distributed systems.
IEEE Software 1(2):19-42, April, 1984.

[5] David R. Cheriton.
VMTP: A Transport Protocol for the Next Generation of Communication Systems.
In *Proceedings of SIGCOMM '86 Symposium on Communications Architectures and Protocols*, pages 406-415. ACM SIGCOMM, Stowe, Vt., August, 1986.

[6] David R. Cheriton and Willy Zwaenepoel.
Distributed process groups in the V kernel.
ACM Transactions on Computer Systems 3(2):77-107, May, 1985.

[7] Communications Machinery Corporation.
DRN-1700 LanScan Ethernet Monitor User's Guide
4th edition, Communications Machinery Corporation, Santa Barbara, California, 1986.

[8] Computer Systems Research Group.
Unix Programmer's Reference Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version
Computer Science Division, University of California at Berkeley, 1986.

[9] *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (Version 1.0)*
Digital Equipment Corporation, Intel, Xerox, 1980.

[10] *TOPS-20 User's Guide*
Digital Equipment Corporation, Maynard, MA., 1980.
Form No. AA-4179C-TM.

[11] *LANalyzer EX 5000E Ethernet Network Analyzer User Manual*
Revision A edition, Excelan, Inc., San Jose, California, 1986.

[12] Ross Finlayson, Timothy Mann, Jeffrey Mogul, Marvin Theimer.
A Reverse Address Resolution Protocol.
RFC 903, Network Information Center, SRI International, June, 1984.

[13] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick.
gprof: a Call Graph Execution Profiler.
In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, pages 120-126. ACM SIGPLAN, June, 1982.

[14] Robert Gurwitz.
Private communication.
1986.

[15] ISO.
ISO Transport Protocol Specification: ISO DP 8073.
RFC 905, Network Information Center, SRI International, April, 1984.

[16] M. Kirk McKusick, Mike Karels, and Sam Leffler.
Performance Improvements and Functional Enhancements in 4.3BSD.
In *Proc. Summer USENIX Conference*, pages 519-531. June, 1985.

[17] Robert. M. Metcalfe and David. R. Boggs.
Ethernet: Distributed packet switching for local computer networks.
Communications of the ACM 19(7):395-404, July, 1976.

[18] *The Sniffer: Operation and Reference Manual*
Network General Corporation, Sunnyvale, California, 1986.

[19] Jon Postel.
Internet Protocol.
RFC 791, Network Information Center, SRI International, September, 1981.

[20] Jon Postel.
Transmission Control Protocol.
RFC 793, Network Information Center, SRI International, September, 1981.

[21] D. M. Ritchie and K. Thompson.
The UNIX timesharing system.
The Bell System Technical Journal 57(6):1905-1929, July/August, 1978.

[22] Sun Microsystems, Inc.
Unix Interface Reference Manual
Sun Microsystems, Inc., Mountain View, California, 1986.
Revision A.

[23] Brent B. Welch.
The Sprite Remote Procedure Call System.
UCB/CSD 86/302, Department of Electrical Engineering and Computer Science, University of California — Berkeley, June, 1986.

Table of Contents

List of Figures

List of Tables