A scalable and explicit event delivery mechanism for UNIX

Gaurav Banga gaurav@netapp.com
Network Appliance Inc., 2770 San Tomas Expressway, Santa Clara, CA 95051
Jeffrey C. Mogul mogul@pa.dec.com
Compaq Computer Corp. Western Research Lab., 250 University Ave., Palo Alto, CA, 94301
Peter Druschel druschel@cs.rice.edu
Department of Computer Science, Rice University, Houston, TX, 77005

Abstract

UNIX applications not wishing to block when doing I/O often use the *select()* system call, to wait for events on multiple file descriptors. The *select()* mechanism works well for small-scale applications, but scales poorly as the number of file descriptors increases. Many modern applications, such as Internet servers, use hundreds or thousands of file descriptors, and suffer greatly from the poor scalability of *select()*. Previous work has shown that while the traditional implementation of *select()* can be improved, the poor scalability is inherent in the design. We present a new event-delivery mechanism, which allows the application to register interest in one or more sources of events, and to efficiently dequeue new events. We show that this mechanism, which requires only minor changes to applications, performs independently of the number of file descriptors.

1 Introduction

An application must often manage large numbers of file descriptors, representing network connections, disk files, and other devices. Inherent in the use of a file descriptor is the possibility of delay. A thread that invokes a blocking I/O call on one file descriptor, such as the UNIX *read()* or *write()* systems calls, risks ignoring all of its other descriptors while it is blocked waiting for data (or for output buffer space).

UNIX supports non-blocking operation for *read()* and *write()*, but a naive use of this mechanism, in which the application polls each file descriptor to see if it might be usable, leads to excessive overheads.

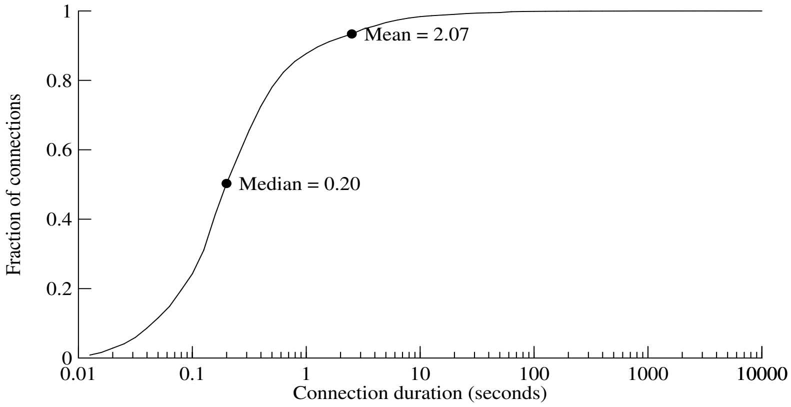
Alternatively, one might allocate a single thread to each activity, allowing one activity to block on I/O without affecting the progress of others. Experience with UNIX and similar systems has shown that this scales badly as the number of threads increases, because of the costs of thread scheduling, context-switching, and thread-state storage space[6, 9]. The use of a single process per connection is even more costly.

The most efficient approach is therefore to allocate a moderate number of threads, corresponding to the amount of available parallelism (for example, one per CPU), and to use non-blocking I/O in conjunction with an efficient mechanism for deciding which descriptors are ready for processing[17]. We focus on the design of this mechanism, and in particular on its efficiency as the number of file descriptors grows very large.

Early computer applications seldom managed many file descriptors. UNIX, for example, originally supported at most 15 descriptors per process[14]. However, the growth of large client-server applications such as database servers, and especially Internet servers, has led to much larger descriptor sets.

Consider, for example, a Web server on the Internet. Typical HTTP mean connection durations have been measured in the range of 2-4 seconds[8, 13]; Figure 1 shows the distribution of HTTP connection durations measured at one of Compaq's firewall proxy servers. Internet connections last so long because of long round-trip times (RTTs), frequent packet loss, and often because of slow (modem-speed) links used for downloading large images or binaries. On the other hand, modern single-CPU servers can handle about 3000 HTTP requests per second[19], and multiprocessors considerably more (albeit in carefully controlled environments). Queueing theory shows that an Internet Web server handling 3000 connections per second, with a mean duration of 2 seconds, will have about 6000 open connections to manage at once (assuming constant interarrival time).

In a previous paper[4], we showed that the BSD UNIX event-notification mechanism, the *select()* system call, scales poorly with increasing connection count. We showed that large connection counts do indeed occur in actual servers, and that the traditional implementation of *select()* could be improved significantly. However, we also found that even our improved *select()* implementation accounts for an unacceptably large share of the overall CPU time. This implies that, no matter how carefully it is implemented, *select()* scales poorly. (Some UNIX systems use a different system call, *poll()*, but we believe that this call has scaling properties at least as bad as those of *select()*, if not worse.)



N = 10,139,681 HTTP connections
Data from 21 October 1998 through 27 October 1998
Fig. 1: Cumulative distribution of proxy connection durations

The key problem with the *select()* interface is that it requires the application to inform the kernel, on each call, of the entire set of "interesting" file descriptors: i.e., those for which the application wants to check readiness. For each event, this causes effort and data motion proportional to the number of interesting file descriptors. Since the number of file descriptors is normally proportional to the event rate, the total cost of *select()* activity scales roughly with the square of the event rate.

In this paper, we explain the distinction between state-based mechanisms, such as *select()*, which check the current status of numerous descriptors, and event-based mechanisms, which deliver explicit event notifications. We present a new UNIX event-based API (application programming interface) that an application may use, instead of *select()*, to wait for events on file descriptors. The API allows an application to register its interest in a file descriptor once (rather than every time it waits for events). When an event occurs on one of these interesting file descriptors, the kernel places a notification on a queue, and the API allows the application to efficiently dequeue event notifications.

We will show that this new interface is simple, easily implemented, and performs independently of the number of file descriptors. For example, with 2000 connections, our API improves maximum throughput by 28%.

2 The problem with *select()*

We begin by reviewing the design and implementation of the *select()* API. The system call is declared as:

```
int select(
        int nfds,
        fd_set *readfds,
        fd_set *writefds,
        fd_set *exceptfds,
        struct timeval *timeout);
```

An *fd_set* is simply a bitmap; the maximum size (in bits) of these bitmaps is the largest legal file descriptor value, which is a system-specific parameter. The *readfds*, *writefds*, and *exceptfds* are in-out arguments, respectively corresponding to the sets of file descriptors that are "interesting" for reading, writing, and exceptional conditions. A given file descriptor might be in more than one of these sets. The *nfds* argument gives the largest bitmap index actually used. The *timeout* argument controls whether, and how soon, *select()* will return if no file descriptors become ready.

Before *select()* is called, the application creates one or more of the *readfds*, *writefds*, or *exceptfds* bitmaps, by asserting bits corresponding to the set of interesting file descriptors. On its return, *select()* overwrites these bitmaps with new values, corresponding to subsets of the input sets, indicating which file descriptors are available for I/O. A member of the *readfds* set is available if there is any available input data; a member of *writefds* is considered writable if the available buffer space exceeds a system-specific parameter (usually 2048 bytes, for TCP sockets). The application then scans the result bitmaps to discover the readable or writable file descriptors, and normally invokes handlers for those descriptors.

Figure 2 is an oversimplified example of how an application typically uses *select()*. One of us has shown[15] that the programming style used here is quite inefficient for large numbers of file descriptors, independent of the problems with *select()*. For example, the construction of the input bitmaps (lines 8 through 12 of Figure 2) should not be done explicitly before each call to *select()*; instead, the application should maintain shadow copies of the input bitmaps, and simply copy these shadows to *readfds* and *writefds*. Also, the scan of the result bitmaps, which are usually quite sparse, is best done word-by-word, rather than bit-by-bit.

Once one has eliminated these inefficiencies, however, *select()* is still quite costly. Part of this cost comes from the use of bitmaps, which must be created, copied into the kernel, scanned by the kernel, subsetted, copied out

```
fd_set readfds, writefds;
struct timeval timeout;
int i, numready;

timeout.tv_sec = 1; timeout.tv_usec = 0;

while (TRUE) {
  FD_ZERO(&readfds); FD_ZERO(&writefds);
  for (i = 0; i <= maxfd; i++) {
    if (WantToReadFD(i)) FD_SET(i, &readfds);
    if (WantToWriteFD(i)) FD_SET(i, &writefds);
  }
  numready = select(maxfd, &readfds,
                    &writefds, NULL, &timeout);
  if (numready < 1) {
    DoTimeoutProcessing();
      continue;
  }

  for (i = 0; i <= maxfd; i++) {
    if (FD_ISSET(i, &readfds)) InvokeReadHandler(i);
    if (FD_ISSET(i, &writefds)) InvokeWriteHandler(i);
  }
}
```

Fig. 2: Simplified example of how *select()* is used

of the kernel, and then scanned by the application. These costs clearly increase with the number of descriptors.

Other aspects of the *select()* implementation also scale poorly. Wright and Stevens provide a detailed discussion of the 4.4BSD implementation[23]; we limit ourselves to a sketch. In the traditional implementation, *select()* starts by checking, for each descriptor present in the input bitmaps, whether that descriptor is already available for I/O. If none are available, then *select()* blocks. Later, when a protocol processing (or file system) module's state changes to make a descriptor readable or writable, that module awakens the blocked process.

In the traditional implementation, the awakened process has no idea which descriptor has just become readable or writable, so it must repeat its initial scan. This is unfortunate, because the protocol module certainly knew what socket or file had changed state, but this information is not preserved. In our previous work on improving *select()* performance[4], we showed that it was fairly easy to preserve this information, and thereby improve the performance of *select()* in the blocking case.

We also showed that one could avoid most of the initial scan by remembering which descriptors had previously been interesting to the calling process (i.e., had been in the input bitmap of a previous *select()* call), and scanning those descriptors only if their state had changed in the interim. The implementation of this technique is somewhat more complex, and depends on set-manipulation operations whose costs are inherently dependent on the number of descriptors.

In our previous work, we tested our modifications using the Digital UNIX V4.0B operating system, and version 1.1.20 of the Squid proxy software[5, 18]. After doing our best to improve the kernel's implementation of *select()*, and Squid's implementation of the procedure that invokes *select()*, we measured the system's performance on a busy non-caching proxy, connected to the Internet and handling over 2.5 million requests/day.

We found that we had approximately doubled the system's efficiency (expressed as CPU time per request), but *select()* still accounted for almost 25% of the total CPU time. Table 1 shows a profile, made with the DCPI [1] tools, of both kernel and user-mode CPU activity during a typical hour of high-load operation.

In the profile *comm_select()*, the user-mode procedure that creates the input bitmaps for *select()* and that scans its output bitmaps, takes only 0.54% of the non-idle CPU time. Some of the 2.85% attributed to *memCopy()* and *memSet()* should also be charged to the creation of the input bitmaps (because the modified Squid uses the shadow-copy method). (The profile also shows a lot of time spent in *malloc()*-related procedures; a future version of Squid will use pre-allocated pools to avoid the overhead of too many calls to *malloc()* and *free()*[22].)

However, the bulk of the *select()*-related overhead is in the kernel code, and accounts for about two thirds of the total non-idle kernel-mode CPU time. Moreover, this measurement reflects a *select()* implementation that we had already improved about as much as we thought possible. Finally, our implementation could not avoid costs dependent on the number of descriptors, implying that the *select()*-related overhead scales worse than linearly. Yet these costs did not seem to be related to intrinsically useful work. We decided to design a scalable replace-

CPU %	Non-idle CPU %	Procedure	Mode
65.43%	100.00%	all non-idle time	kernel
34.57%		all idle time	kernel
16.02%	24.49%	all select functions	kernel
9.42%	14.40%	*select*	kernel
3.71%	5.67%	*new_soo_select*	kernel
2.82%	4.31%	*new_selscan_one*	kernel
0.03%	0.04%	*new_undo_scan*	kernel
15.45%	23.61%	*malloc*-related code	user
4.10%	6.27%	*in_pcblookup*	kernel
2.88%	4.40%	all TCP functions	kernel
0.94%	1.44%	*memCopy*	user
0.92%	1.41%	*memset*	user
0.88%	1.35%	*bcopy*	kernel
0.84%	1.28%	*read_io_port*	kernel
0.72%	1.10%	*_doprnt*	user
0.36%	0.54%	*comm_select*	user

Profile on 1998-09-09 from 11:00 to 12:00 PDT
mean load = 56 requests/sec.
peak load ca. 131 requests/sec

Table 1: Profile - modified kernel, Squid on live proxy

ment for *select()*.

2.1 The *poll()* system call

In the System V UNIX environment, applications use the *poll()* system call instead of *select()*. This call is declared as:

```
struct pollfd {
      int     fd;
      short   events;
      short   revents;
};

int poll(
        struct pollfd filedes[];
        unsigned int nfds;
        int timeout   /* in milliseconds */);
```

The *filedes* argument is an in-out array with one element for each file descriptor of interest; *nfds* gives the array length. On input, the *events* field of each element tells the kernel which of a set of conditions are of interest for the associated file descriptor *fd*. On return, the *revents* field shows what subset of those conditions hold true. These fields represent a somewhat broader set of conditions than the three bitmaps used by *select()*.

The *poll()* API appears to have two advantages over *select()*: its array compactly represents only the file descriptors of interest, and it does not destroy the input fields of its in-out argument. However, the former advantage is probably illusory, since *select()* only copies 3 bits per file descriptor, while *poll()* copies 64 bits. If the number of interesting descriptors exceeds 3/64 of the highest-numbered active file descriptor, *poll()* does more copying than *select()*. In any event, it shares the same scaling problem, doing work proportional to the number of interesting descriptors rather than constant effort, per event.

3 Event-based vs. state-based notification mechanisms

Recall that we wish to provide an application with an efficient and scalable means to decide which of its file descriptors are ready for processing. We can approach this in either of two ways:

1. A *state-based* view, in which the kernel informs the application of the current state of a file descriptor (e.g., whether there is any data currently available for reading).

2. An *event-based* view, in which the kernel informs the application of the occurrence of a meaningful event for a file descriptor (e.g., whether new data has been added to a socket's input buffer).

The *select()* mechanism follows the state-based approach. For example, if *select()* says a descriptor is ready for reading, then there is data in its input buffer. If the application reads just a portion of this data, and then calls *select()* again before more data arrives, *select()* will again report that the descriptor is ready for reading.

The state-based approach inherently requires the kernel to check, on every notification-wait call, the status of each member of the set of descriptors whose state is being tested. As in our improved implementation of *select()*, one can elide part of this overhead by watching for events that change the state of a descriptor from unready to ready. The kernel need not repeatedly re-test the state of a descriptor known to be unready.

However, once *select()* has told the application that a descriptor is ready, the application might or might not perform operations to reverse this state-change. For example, it might not read anything at all from a ready-for-reading input descriptor, or it might not read all of the pending data. Therefore, once *select()* has reported that a descriptor is ready, it cannot simply ignore that descriptor on future calls. It must test that descriptor's state, at least until it becomes unready, even if no further I/O events occur. Note that elements of *writefds* are usually ready.

Although *select()* follows the state-based approach, the kernel's I/O subsystems deal with events: data packets arrive, acknowledgements arrive, disk blocks arrive, etc. Therefore, the *select()* implementation must transform notifications from an internal event-based view to an external state-based view. But the "event-driven" applications that use *select()* to obtain notifications ultimately follow the event-based view, and thus spend effort tranforming information back from the state-based model. These dual transformations create extra work.

Our new API follows the event-based approach. In this model, the kernel simply reports a stream of events to the application. These events are monotonic, in the sense that they never decrease the amount of readable data (or writable buffer space) for a descriptor. Therefore, once an event has arrived for a descriptor, the application can either process the descriptor immediately, or make note of the event and defer the processing. The kernel does not track the readiness of any descriptor, so it does not perform work proportional to the number of descriptors; it only performs work proportional to the number of events.

Pure event-based APIs have two problems:

1. Frequent event arrivals can create excessive communication overhead, especially for an application that is not interested in seeing every individual event.

2. If the API promises to deliver information about each individual event, it must allocate storage proportional to the event rate.

Our API does not deliver events asynchronously (as would a signal-based mechanism; see Section 8.2), which helps to eliminate the first problem. Instead, the API allows an application to efficiently discover descriptors that have had event arrivals. Once an event has arrived for a descriptor, the kernel coalesces subsequent event arrivals for that descriptor until the application learns of the first one; this reduces the communication rate, and avoids the need to store per-event information. We believe that most applications do not need explicit per-event information, beyond that available in-band in the data stream.

By simplifying the semantics of the API (compared to *select()*), we remove the necessity to maintain information in the kernel that might not be of interest to the application. We also remove a pair of transformations between the event-based and state-based views. This improves the scalability of the kernel implementation, and leaves the application sufficient flexibility to implement the appropriate event-management algorithms.

4 Details of the programming interface

An application might not be always interested in events arriving on all of its open file descriptors. For example, as mentioned in Section 8.1, the Squid proxy server temporarily ignores data arriving in dribbles; it would rather process large buffers, if possible.

Therefore, our API includes a system call allowing a thread to declare its interest (or lack of interest) in a file descriptor:

```
#define    EVENT_READ        0x1
#define    EVENT_WRITE       0x2
#define    EVENT_EXCEPT      0x4

int declare_interest(int fd,
                     int interestmask,
                     int *statemask);
```

The thread calls this procedure with the file descriptor in question. The *interestmask* indicate whether or not the thread is interested in reading from or writing to the descriptor, or in exception events. If *interestmask* is zero, then the thread is no longer interested in any events for the descriptor. Closing a descriptor implicitly removes any declared interest.

Once the thread has declared its interest, the kernel tracks event arrivals for the descriptor. Each arrival is added to a per-thread queue. If multiple threads are interested in a descriptor, a per-socket option selects between two ways to choose the proper queue (or queues). The default is to enqueue an event-arrival record for each interested thread, but by setting the SO_WAKEUP_ONE flag, the application indicates that it wants an event arrival delivered only to the first eligible thread.

If the *statemask* argument is non-NULL, then *declare_interest()* also reports the current state of the file descriptor. For example, if the EVENT_READ bit is set in this value, then the descriptor is ready for reading. This feature avoids a race in which a state change occurs after the file has been opened (perhaps via an *accept()* system call) but before *declare_interest()* has been called. The implementation guarantees that the *statemask* value reflects the descriptor's state before any events are added to the thread's queue. Otherwise, to avoid missing any events, the application would have to perform a non-blocking *read* or *write* after calling *declare_interest()*.

To wait for additional events, a thread invokes another new system call:

```
typedef struct {
  int fd;
  unsigned mask;
} event_descr_t;

int get_next_event(int array_max,
                   event_descr_t *ev_array,
                   struct timeval *timeout);
```

The *ev_array* argument is a pointer to an array, of length *array_max*, of values of type *event_descr_t*. If any events are pending for the thread, the kernel dequeues, in FIFO order, up to *array_max* events[1]. It reports these dequeued events in the *ev_array* result array. The *mask* bits in each *event_descr_t* record, with the same definitions as used in *declare_interest()*, indicate the current

[1] A FIFO ordering is not intrinsic to the design. In another paper[3], we describe a new kernel mechanism, called *resource containers*, which allows an application to specify the priority in which the kernel enqueues events.

state of the corresponding descriptor *fd*. The function return value gives the number of events actually reported.

By allowing an application to request an arbitrary number of event reports in one call, it can amortize the cost of this call over multiple events. However, if at least one event is queued when the call is made, it returns immediately; we do not block the thread simply to fill up its *ev_array*.

If no events are queued for the thread, then the call blocks until at least one event arrives, or until the timeout expires.

Note that in a multi-threaded application (or in an application where the same socket or file is simultaneously open via several descriptors), a race could make the descriptor unready before the application reads the *mask* bits. The application should use non-blocking operations to read or write these descriptors, even if they appear to be ready. The implementation of *get_next_event()* does attempt to try to report the current state of a descriptor, rather than simply reporting the most recent state transition, and internally suppresses any reports that are no longer meaningful; this should reduce the frequency of such races.

The implementation also attempts to coalesce multiple reports for the same descriptor. This may be of value when, for example, a bulk data transfer arrives as a series of small packets. The application might consume all of the buffered data in one system call; it would be inefficient if the application had to consume dozens of queued event notifications corresponding to one large buffered read. However, it is not possible to entirely eliminate duplicate notifications, because of races between new event arrivals and the *read*, *write*, or similar system calls.

5 Use of the programming interface

Figure 3 shows a highly simplified example of how one might use the new API to write parts of an event-driven server. We omit important details such as error-handling, multi-threading, and many procedure definitions.

The *main_loop()* procedure is the central event dispatcher. Each iteration starts by attempting to dequeue a batch of events (here, up to 64 per batch), using *get_next_event()* at line 9. If the system call times out, the application does its timeout-related processing. Otherwise, it loops over the batch of events, and dispatches event handlers for each event. At line 16, there is a special case for the socket(s) on which the application is listening for new connections, which is handled differently from data-carrying sockets.

We show only one handler, for these special listen-sockets. In initialization code not shown here, these listen-sockets have been set to use the non-blocking option. Therefore, the *accept()* call at line 30 will never block, even if a race with the *get_next_event()* call somehow causes this code to run too often. (For example, a remote client might close a new connection before we have a chance to accept it.) If *accept()* does successfully return the socket for a new connection, line 31 sets it to use non-blocking I/O. At line 32, *declare_interest()* tells the kernel that the application wants to know about future read and write events. Line 34 tests to see if any data became available before we called *declare_interest()*; if so, we read it immediately.

6 Implementation

We implemented our new API by modifying Digital UNIX V4.0D. We started with our improved *select()* implementation [4], reusing some data structures and support functions from that effort. This also allows us to measure our new API against the best known *select()* implementation without varying anything else. Our current implementation works only for sockets, but could be extended to other descriptor types. (References below to the "protocol stack" would then include file system and device driver code.)

For the new API, we added about 650 lines of code. The *get_next_event()* call required about 320 lines, *declare_interest()* required 150, and the remainder covers changes to protocol code and support functions. In contrast, our previous modifications to *select()* added about 1200 lines, of which we reused about 100 lines in implementing the new API.

For each application thread, our code maintains four data structures. These include INTERESTED.read, INTERESTED.write, and INTERESTED.except, the sets of descriptors designated via *declare_interest()* as "interesting" for reading, writing, and exceptions, respectively. The other is HINTS, a FIFO queue of events posted by the protocol stack for the thread.

A thread's first call to *declare_interest()* causes creation of its INTERESTED sets; the sets are resized as necessary when descriptors are added. The HINTS queue is created upon thread creation. All four sets are destroyed when the thread exits. When a descriptor is closed, it is automatically removed from all relevant INTERESTED sets.

Figure 4 shows the kernel data structures for an example in which a thread has declared read interest in descriptors 1 and 4, and write interest in descriptor 0. The three INTERESTED sets are shown here as one-byte bitmaps, because the thread has not declared interest in any higher-numbered descriptors. In this example, the HINTS queue for the thread records three pending events, one each for descriptors 1, 0, and 4.

A call to *declare_interest()* also adds an element to the corresponding socket's "reverse-mapping" list; this element includes both a pointer to the thread and the descriptor's index number. Figure 5 shows the kernel

```
#define MAX_EVENTS 64
struct event_descr_t event_array[MAX_EVENTS];

main_loop(struct timeval timeout)
{
  int i, n;

  while (TRUE) {
    n = get_next_event(MAX_EVENTS, &event_array, &timeout);
    if (n < 1) {
      DoTimeoutProcessing(); continue;
    }

    for (i = 0; i < n; i++) {
      if (event_array[i].mask & EVENT_READ)
        if (ListeningOn(event_array[i].fd))
          InvokeAcceptHandler(event_array[i].fd);
        else
          InvokeReadHandler(event_array[i].fd);
      if (event_array[i].mask & EVENT_WRITE)
        InvokeWriteHandler(event_array[i].fd);
    }
  }
}

InvokeAcceptHandler(int listenfd)
{
  int newfd, statemask;

  while ((newfd = accept(listenfd, NULL, NULL)) >= 0) {
    SetNonblocking(newfd);
    declare_interest(newfd, EVENT_READ|EVENT_WRITE,
                     &statemask);
    if (statemask & EVENT_READ)
      InvokeReadHandler(newfd);
  }
}
```

Fig. 3: Simplified example of how the new API might be used

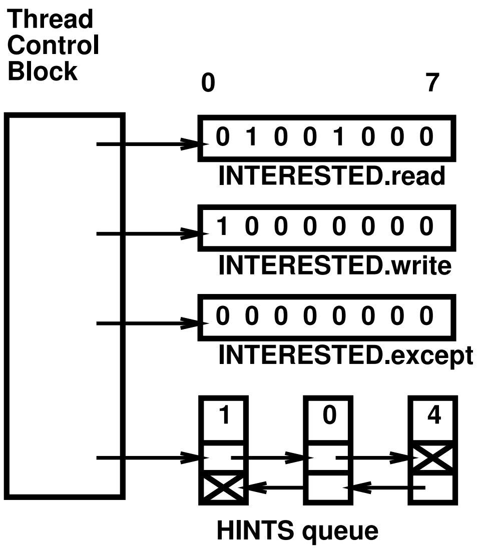


Fig. 4: Per-thread data structures

data structures for an example in which Process 1 and Process 2 hold references to Socket A via file descriptors 2 and 4, respectively. Two threads of Process 1 and one thread of Process 2 are interested in Socket A, so the reverse-mapping list associated with the socket has pointers to all three threads.

When the protocol code processes an event (such as data arrival) for a socket, it checks the reverse-mapping list. For each thread on the list, if the index number is found in the thread's relevant INTERESTED set, then a notification element is added to the thread's HINTS queue.

To avoid the overhead of adding and deleting the reverse-mapping lists too often, we never remove a reverse-mapping item until the descriptor is closed. This means that the list is updated at most once per descriptor lifetime. It does add some slight per-event overhead for a socket while a thread has revoked its interest in that descriptor; we believe this is negligible.

We attempt to coalesce multiple event notifications for a single descriptor. We use another per-thread bitmap, in-

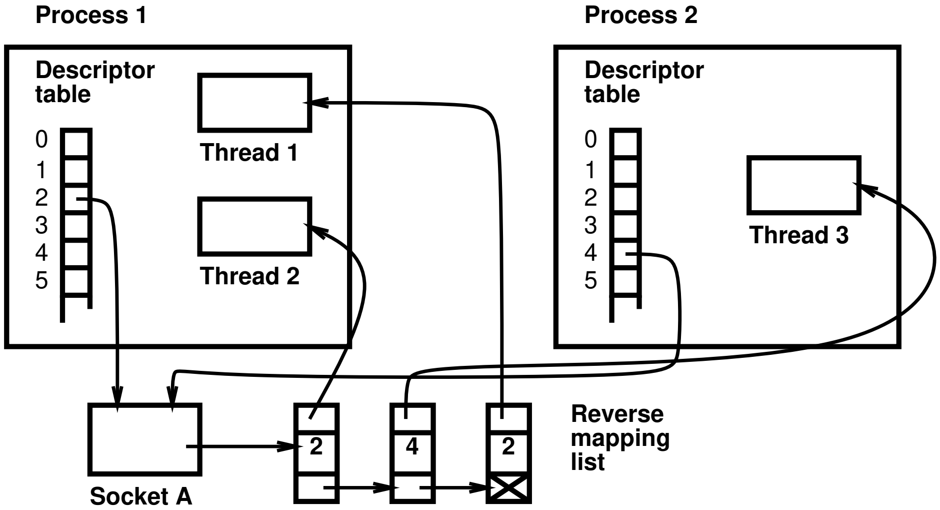


Fig. 5: Per-socket data structures

dexed by file descriptor number, to note that the HINTS queue contains a pending element for the descriptor. The protocol code tests and sets these bitmap entries; they are cleared once *get_next_event()* has delivered the corresponding notification. Thus, N events on a socket between calls to *get_next_event()* lead to just one notification.

Each call to *get_next_event()*, unless it times out, dequeues one or more notification elements from the HINTS queue in FIFO order. However, the HINTS queue has a size limit; if it overflows, we discard it and deliver events in descriptor order, using a linear search of the INTERESTED sets – we would rather deliver things in the wrong order than block progress. This policy could lead to starvation, if the *array_max* parameter to *get_next_event()* is less than the number of descriptors, and may need revision.

We note that there are other possible implementations for the new API. For example, one of the anonymous reviewers suggested using a linked list for the per-thread queue of pending events, reserving space for one list element in each socket data structure. This approach seems to have several advantages when the SO_WAKEUP_ONE option is set, but might not be feasible when each event is delivered to multiple threads.

7 Performance

We measured the performance of our new API using a simple event-driven HTTP proxy program. This proxy does not cache responses. It can be configured to use either *select()* or our new event API.

In all of the experiments presented here, we generate load using two kinds of clients. The "hot" connections come from a set of processes running the S-Client software [2], designed to generate realistic request loads, characteristic of WAN clients. As in our earlier work [4], we also use a *load-adding client* to generate a large number of "cold" connections: long-duration dummy connections that simulate the effect of large WAN delays. The load-adding client process opens as many as several thousand connections, but does not actually send any requests. In essence, we simulate a load with a given arrival rate and duration distribution by breaking it into two pieces: S-Clients for the arrival rate, and load-adding clients for the duration distribution.

The proxy relays all requests to a Web server, a single-process event-driven program derived from thttpd [20], with numerous performance improvements. (This is an early version of the Flash Web server [17].) We take care to ensure that the clients, the Web server, and the network itself are never bottlenecks. Thus, the proxy server system is the bottleneck.

7.1 Experimental environment

The system under test, where the proxy server runs, is a 500MHz Digital Personal Workstation (Alpha 21164, 128MB RAM, SPECint95 = 15.7), running our modified version of Digital UNIX V4.0D. The client processes run on four identical 166Mhz Pentium Pro machines (64MB RAM, FreeBSD 2.2.6). The Web server program runs on a 300 MHz Pentium II (128MB RAM, FreeBSD 2.2.6).

A switched full-duplex 100 Mbit/sec Fast Ethernet connects all machines. The proxy server machine has two network interfaces, one for client traffic and one for Web-server traffic.

7.2 API function costs

We performed experiments to find the basic costs of our new API calls, measuring how these costs scale with the number of connections per process. Ideally, the costs should be both low and constant.

In these tests, S-Client software simulates HTTP clients generating requests to the proxy. Concurrently, a load-adding client establishes some number of cold connections to the proxy server. We started measurements only after a dummy run warmed the Web server's file cache. During these measurements, the proxy's CPU is saturated, and the proxy application never blocks in *get_next_event()*; there are always events queued for delivery.

The proxy application uses the Alpha's cycle counter to measure the elapsed time spent in each system call; we report the time averaged over 10,000 calls.

To measure the cost of *get_next_event()*, we used S-Clients generating requests for a 40 MByte file, thus causing thousands of events per connection. We ran trials with *array_max* (the maximum number of events delivered per call) varying between 1 and 10; we also varied the number of S-Client processes. Figure 6 shows that the cost per call, with 750 cold connections, varies linearly with *array_max*, up to a point limited (apparently) by the concurrency of the S-Clients.

For a given *array_max* value, we found that varying the number of cold connections between 0 and 2000 has almost no effect on the cost of *get_next_event()*, accounting for variation of at most 0.005% over this range.

We also found that increasing the hot-connection rate did not appear to increase the per-event cost of *get_next_event()*. In fact, the event-batching mechanism reduces the per-event cost, as the proxy falls further behind. The cost of all event API operations in our implementation is independent of the event rate, as long as the maximum size of the HINTS queue is configured large enough to hold one entry for each descriptor of the process.

To measure the cost of the *declare_interest()* system call, we used 32 S-Clients making requests for a 1 KByte file. We made separate measurements for the "declaring interest" case (adding a new descriptor to an INTERESTED set) and the "revoking interest" case (removing a descriptor); the former case has a longer code path. Figure 7 shows slight cost variations with changes in the number of cold connections, but these may be measurement artifacts.

7.3 Proxy server performance

We then measured the actual performance of our simple proxy server, using either *select()* or our new API. In these experiments, all requests are for the same (static) 1 Kbyte file, which is therefore always cached in the Web server's memory. (We ran additional tests using 8 Kbyte files; space does not permit showing the results, but they display analogous behavior.)

In the first series of tests, we always used 32 hot connections, but varied the number of cold connections between 0 and 2000. The hot-connection S-Clients are configured to generate requests as fast as the proxy system can handle; thus we saturated the proxy, but never overloaded it. Figure 8 plots the throughput achieved for three kernel configurations: (1) the "classical" implementation of *select()*, (2) our improved implementation of *select()*, and (3) the new API described in this paper. All kernels use a scalable version of the *ufalloc()* file-descriptor allocation function [4]; the normal version does not scale well. The results clearly indicate that our new API performs independently of the number of cold connections, while *select()* does not. (We also found that the proxy's throughput is independent of *array_max*.)

In the second series of tests, we fixed the number of cold connections at 750, and measured response time (as seen by the clients). Figure 9 shows the results. When using our new API, the proxy system exhibits much lower latency, and saturates at a somewhat higher request load (1348 requests/sec., vs. 1291 request/sec. for the improved *select()* implementation).

Table 2 shows DCPI profiles of the proxy server in the three kernel configurations. These profiles were made using 750 cold connections, 50 hot connections, and a total load of 400 requests/sec. They show that the new event API significantly increases the amount of CPU idle time, by almost eliminating the event-notification overhead. While the classical *select()* implementation consumes 34% of the CPU, and our improved *select()* implementation consumes 12%, the new API consumes less than 1% of the CPU.

8 Related work

To place our work in context, we survey other investigations into the scalability of event-management APIs, and the design of event-management APIs in other operating systems.

8.1 Event support in NetBIOS and Win32

The NetBIOS interface[12] allows an application to wait for incoming data on multiple network connections. NetBIOS does not provide a procedure-call interface; instead, an application creates a "Network Control Block" (NCB), loads its address into specific registers, and then invokes NetBIOS via a software interrupt. NetBIOS provides a command's result via a callback.

The NetBIOS "receive any" command returns (calls back) when data arrives on any network "session" (connection). This allows an application to wait for arriving data on an arbitrary number of sessions, without having to enumerate the set of sessions. It does not appear possible to wait for received data on a subset of the active sessions.

The "receive any" command has numerous limitations, some of which are the result of a non-extensible design. The NCB format allows at most 254 sessions, which obviates the need for a highly-scalable implement-

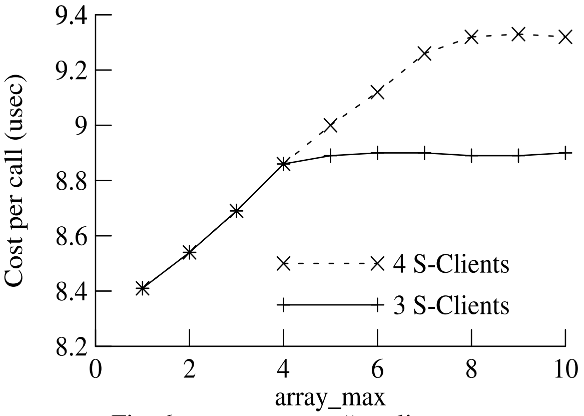


Fig. 6: *get_next_event()* scaling

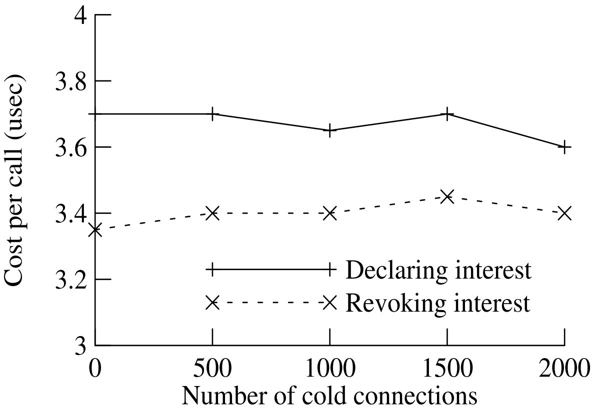


Fig. 7: *declare_interest()* scaling

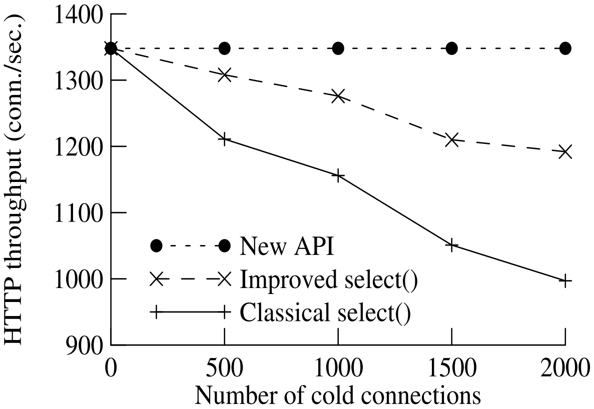


Fig. 8: HTTP rate vs. cold connections

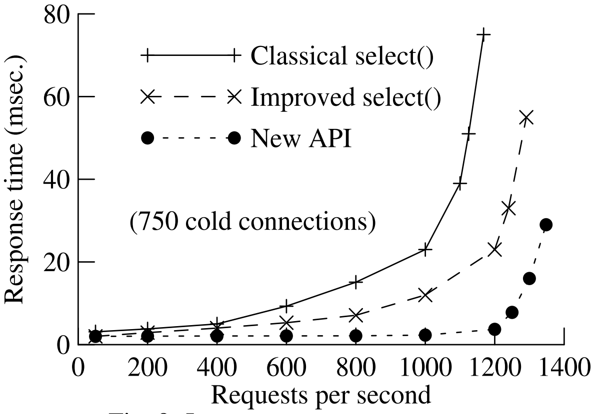


Fig. 9: Latency vs. request rate

ation. The command does not allow an application to discover when a once-full output buffer becomes writable, nor does it apply to disk files.

In the Win32 programming environment[10], the use of NetBIOS is strongly discouraged. Win32 includes a procedure named *WaitForMultipleObjects()*, declared as:

```
DWORD WaitForMultipleObjects(
  DWORD cObjects,
    // number of handles in handle array
  CONST HANDLE * lphObjects,
    // address of object-handle array
  BOOL fWaitAll,
    // flag: wait for all or for just one
  DWORD dwTimeout
    // time-out interval in milliseconds
  );
```

This procedure takes an array of Win32 objects (which could include I/O handles, threads, processes, mutexes, etc.) and waits for either one or all of them to complete. If the *fWaitAll* flag is FALSE, then the returned value is the array index of the ready object. It is not possible to learn about multiple objects in one call, unless the application is willing to wait for completion on all of the listed objects.

This procedure, like *select()*, might not scale very well to large numbers of file handles, for a similar reason: it passes information about all potential event sources every time it is called. (In any case, the object-handle array may contain no more than 64 elements.) Also, since *WaitForMultipleObjects* must be called repeatedly to obtain multiple events, and the array is searched linearly, a frequent event rate on objects early in the array can starve service for higher-indexed objects.

Windows NT 3.5 added a more advanced mechanism for detecting I/O events, called an I/O completion port (IOCP)[10, 21]. This ties together the threads mechanism with the I/O mechanism. An application calls *CreateIoCompletionPort()* to create an IOCP, and then makes an additional call to *CreateIoCompletionPort()* to associate each interesting file handle with that IOCP. Each such call also provides an application-specified "CompletionKey" value that will be associated with the file handle.

An application thread waits for I/O completion events using the *GetQueuedCompletionStatus()* call:

```
BOOL GetQueuedCompletionStatus(
  HANDLE CompletionPort,
  LPDWORD lpNumberOfBytesTransferred,
  LPDWORD CompletionKey,
  LPOVERLAPPED *lpOverlapped,
  DWORD dwMillisecondTimeout);
```

Upon return, the *CompletionKey* variable holds the

Classical select() CPU %	Scalable select() CPU %	New event API CPU %	Procedure	Mode
18.09%	33.01%	59.01%	all idle time	kernel
33.51%	*12.02%*	*0.68%*	*all kernel* select *or event functions*	kernel
13.78%	N.A.	N.A.	*soo_select()*	kernel
9.11%	N.A.	N.A.	*selscan()*	kernel
8.40%	N.A.	N.A.	*undo_scan()*	kernel
2.22%	12.02%	N.A.	*select()*	kernel
N.A.	0.57%	N.A.	*new_soo_select()*	kernel
N.A.	0.47%	N.A.	*new_selscan_one()*	kernel
N.A.	N.A.	0.40%	*get_next_event()*	kernel
N.A.	N.A.	0.15%	*declare_interest()*	kernel
N.A.	N.A.	0.13%	*revoke_interest()*	kernel
2.01%	1.95%	1.71%	*_Xsyscall()*	kernel
1.98%	1.88%	1.21%	*main()*	user
1.91%	1.90%	1.69%	*_doprnt()*	user
1.63%	1.58%	1.54%	*memset()*	user
1.29%	1.31%	1.47%	*read_io_port()*	kernel
1.11%	1.15%	1.20%	*syscall()*	kernel
1.09%	1.11%	1.11%	*_XentInt()*	kernel
1.08%	1.06%	1.19%	*malloc()*	kernel

750 cold connections, 50 hot connections, 400 requests/second, 1KB/request

Table 2: Effect of event API on system CPU profile

value associated, via *CreateIoCompletionPort()*, with the corresponding file handle. Several threads might be blocked in this procedure waiting for completion events on the same IOCP. The kernel delivers the I/O events in FIFO order, but selects among the blocked threads in LIFO order, to reduce context-switching overhead.

The IOCP mechanism seems to have no inherent limits on scaling to large numbers of file descriptors or threads. We know of no experimental results confirming its scalability, however.

Once a handle has been associated with an IOCP, there is no way to disassociate it, except by closing the handle. This somewhat complicates the programmer's task; for example, it is unsafe to use as the CompletionKey the address of a data structure that might be reallocated when a file handle is closed. Instead, the application should use a nonce value, implying another level of indirection to obtain the necessary pointer. And while the application might use several IOCPs to segregate file handles into different priority classes, it cannot move a file handle from one IOCP to another as a way of adjusting its priority.

Some applications, such as the Squid proxy[5, 18], temporarily ignore I/O events on an active file descriptor, to avoid servicing data arriving as a lengthy series of small dribbles. This is easily done with the UNIX *select()* call, by removing that descriptor from the input bitmap; it is not clear if this can be done using an IOCP.

Hu et al.[11] discuss several different NT event dispatching and concurrency models in the context of a Web server, and show how the server's performance varies according to the model chosen. However, they did not measure how performance scales with large numbers of open connections, but limited their measurements to at most 16 concurrent clients.

In summary, the IOCP mechanism in Windows NT is similar to the API we propose for UNIX, and predates our design by several years (although we were initially unaware of it). The differences between the designs may or may not be significant; we look forward to a careful analysis of IOCP performance scaling. Our contribution is not the concept of a pending-event queue, but rather its application to UNIX, and our quantitative analysis of its scalability.

8.2 Queued I/O completion signals in POSIX

The POSIX[16] API allows an application to request the delivery of a signal (software interrupt) when I/O is possible for a given file descriptor. The POSIX Realtime Signals Extension allows an application to request that delivered signals be queued, and that the signal handler be invoked with a parameter giving the associated file descriptor. The combination of these facilities provides a scalable notification mechanism.

We see three problems that discourage the use of signals. First, signal delivery is more expensive than the specialized event mechanism we propose. On our test system, signal delivery (for SIGIO) requires 10.7 usec, versus about 8.4 usec for *get_next_event()* (see figure 6), and (unlike *get_next_event()*) the signal mechanism cannot batch notifications for multiple descriptors in one invocation. Second, asynchronous invocation of handlers implies the use of some sort of locking mechanism, which adds overhead and complexity. Finally, the use of signals eliminates application control over which thread is invoked.

8.3 Port sets in Mach

The Mach operating system[24] depends on message-based communication, using "ports" to represent message end-points. Ports are protected with capability-like "send rights" and "receive rights." All system operations are performed using messages; for example, virtual memory faults are converted into messages sent to the backing-store port of the associated memory object. Other communication models, such as TCP byte streams, are constructed on top of this message-passing layer.

Each port has a queue of pending messages. A thread may use the *msg_receive()* system call to retrieve a message from the queue of a single port, or wait for a message to arrive if the queue is empty.

A thread with receive rights for many ports may create a "port set", a first-class object containing an arbitrary subset of these receive rights[7]. The thread may then invoke *msg_receive()* on that port set (rather than on the underlying ports), receiving messages from all of the contained ports in FIFO order. Each message is marked with the identity of the original receiving port, allowing the thread to demultiplex the messages. The port set approach scales efficiently: the time required to retrieve a message from a port set should be independent of the number of ports in that set.

Port sets are appropriate for a model in which all communication is done with messages, and in which the system provides the necessary facilities to manage message ports (not necessarily a simple problem[7]). Introducing port sets into UNIX, where most communication follows a byte-stream model, might require major changes to applications and existing components.

9 Future work

The *select()* mechanism can be confusing in multithreaded programs, especially on multiprocessors. Because *select()* returns the state of a descriptor, instead of an event notification, two threads blocked in *select()* could awaken at the same time, and would need additional synchronization to avoid handling the same descriptor. Our event-based API should make writing threaded applications more natural, because (with the SO_WAKEUP_ONE option described in Section 4) it delivers each event at most once. We have not yet explored this area in detail.

Our existing API requires each thread in a process to call *declare_interest()* for each descriptor that it is interested in. This requirement might add excessive overhead for a multi-threaded program using a large pool of interchangeable worker threads. We could augment the API with another system call:

```
int declare_processwide_interest(int fd,
                     int interestmask,
                     int *statemask);
```

The result of this system call would be the equivalent of invoking *declare_interest()* in every existing and future thread of the calling process. (It might also implicitly set SO_WAKEUP_ONE for the descriptor.) After this call, any thread of the process could wait for events on this descriptor using *get_next_event()*.

An application handling thousands of descriptors might want to set event-delivery priorities, to control the order in which the kernel delivers events. In another paper [3], we introduced the *resource container* abstraction, which (among other benefits) allows an application to set kernel-level priorities for descriptor processing. In that paper we showed how an event-based API, such as the one presented here, is a useful component of end-to-end priority control in networked applications. We look forward to gaining experience with the combination of priority control and an event-based API in complex applications.

10 Summary

We showed that the scalability of an operating system's event notification mechanism has a direct effect on application performance scalability. We also showed that the *select()* API has inherently poor scalability, but that it can be replaced with a simple event-oriented API. We implemented this API and showed that it does indeed improve performance on a real application.

11 Acknowledgments

We would like to thank Mitch Lichtenberg for helping us understand the NetBIOS, Win32, and Windows NT event-management APIs. We would also like to thank the anonymous reviewers for their suggestions.

This work was done while Gaurav Banga was a student at Rice University. It was supported in part by NSF Grants CCR-9803673 and CCR-9503098, by Texas TATP Grant 003604, and by an equipment loan by the Digital Equipment Corporation subsidiary of Compaq Computer Corporation.

References

[1] J. Anderson, L. M. Berc, et al. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 1–14, San Malo, France, Oct. 1997.

[2] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, pages 61–71, Monterey, CA, Dec. 1997.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. 3rd. Symp. on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.

[4] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. 1998 USENIX Annual Technical Conf.*, pages 1–12, New Orleans, LA, June 1998. USENIX.

[5] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, pages 153–163, San Diego, CA, Jan. 1996.

[6] J. B. Chen. Personal communication, Nov. 1998.

[7] R. P. Draves. A Revised IPC Interface. In *Proc. USENIX Mach Conference*, pages 101–122, Burlington, VT, October 1990.

[8] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proc. USENIX Symposium on Internet Technologies and Systems*, pages 207–218, Monterey, CA, December 1997.

[9] D. Grunwald. Personal communication, Mar. 1998.

[10] J. M. Hart. *Win32 System Programming*. Addison Wesley Longman, Reading, MA, 1997.

[11] J. Hu, I. Pyarali, and D. C. Schmidt. Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks. In *Proc. Global Internet Conference (held as part of GLOBECOM '97)*, Phoenix, AZ, November 1997.

[12] IBM. *Local Area Network Technical Reference*. Research Triangle Park, NC, 4th edition, 1990.

[13] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. Symposium on Internet Technologies and Systems*, pages 13–22, Monterey, CA, December 1997. USENIX.

[14] J. Lions. *Lion's Commentary on UNIX 6th Edition with Source Code*. Peer-to-Peer Communications, San Jose, CA, 1996.

[15] J. C. Mogul. Speedier Squid: A case study of an Internet server performance problem. *;login:*, pages 50–58, February 1999.

[16] The Open Group, Woburn, MA. *The Single UNIX Specification, Version 2 - 6 Vol Set for UNIX 98*, 1997.

[17] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proc. 1999 USENIX Annual Technical Conf.*, Monterey, CA, June 1998. USENIX.

[18] Squid. http://squid.nlanr.net/Squid/.

[19] The Standard Performance Evaluation Corporation (SPEC). SPECweb96 Results. http://www.specbench.org/osg/web96/results/.

[20] thttpd. http://www.acme.com/software/thttpd/.

[21] J. Vert. Writing Scalable Applications for Windows NT. http://www.microsoft.com/win32dev/base/SCALABIL.HTM, 1995.

[22] D. Wessels. Personal communication, September 1998.

[23] G. Wright and W. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.

[24] M. Young, A. Tevanian, Jr., R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proc. 11th Symposium on Operating Systems Principles*, pages 63–76, Austin, TX, November 1987.