Scalability of Linux Event-Dispatch Mechanisms

Abhishek Chandra, David Mosberger
Internet and Mobile Systems Laboratory
HP Laboratories Palo Alto
HPL-2000-174
December 14th, 2000*

E-mail: abhishek@cs.umass.edu, davidm@hplhp.com

event dispatching, Web servers, Internet servers, Linux, performance

Many Internet servers these days have to handle not just heavy request loads, but also increasingly face large numbers of concurrent connections. In this paper, we discuss some of the event-dispatch mechanisms used by Internet servers to handle the network I/O generated by these request loads. We focus on the mechanisms supported by the Linux kernel, and measure their performance in terms of their dispatch overhead and dispatch throughput. Our comparative studies show that POSIX.4 Real Time signals (RT signals) are a highly efficient mechanism in terms of the overhead and also provide good throughput compared to mechanisms like select() and /dev/poll. We also look at some limitations of RT signals and propose an enhancement to the default RT signal implementation which we call signal-per-fd. This enhancement has the advantage of significantly reducing the complexity of a server implementation, increasing its robustness under high load, and also potentially increasing its throughput. In addition, our results also show that, contrary to conventional wisdom, even a select()-based server can provide high throughput if its overhead is amortized by performing more useful work per select() call.

* Internal Accession Date Only
Approved for External Publication

1 Introduction

The fast growth of the Web and e-commerce has led to a large increase in Internet traffic. Most network applications such as Web servers and proxies have to handle heavy loads from clients spread all across the globe. In addition to high request rates, servers also have to handle a large number of concurrent connections, many of which are idle most of the time. This is because the connection times are large due to (i) the "last-mile problem" [3], which has the effect that most clients connect to the Internet through slow modems, and (ii) due to the geographically distributed nature of the Internet, which causes much of the traffic to travel across many hops, increasing both latency and the probability of packet drops due to congestion. For Web servers, the problem of long connections is exacerbated by the HTTP/1.1 protocol [5], which provides for persistent TCP connections that can be reused to handle multiple interactions with the server. These persistent connections further add to the length of the connection times. The bottom line is that servers need to service the high incoming request load, while simultaneously handling a large number of concurrent connections efficiently.

To handle these demands, many high-performance Web servers are structured as event-handling applications [9, 14, 16]. These servers employ event-dispatch mechanisms provided by the underlying operating system to handle the network I/O on multiple concurrent connections. Some studies have looked at the scalability issues of some of these mechanisms and found that traditional dispatch mechanisms are not very scalable [1]. While the performance of Web servers clearly is important, we should not forget that there are many other Internet services, such as ftp servers, proxy caches, and mail servers, that have to deal with similar scalability concerns. For example, poor scalability is one of the primary reasons the number of concurrent connections on many ftp servers is limited to a small number (around 30-50).

Another approach to building Internet servers that can handle high request loads and large number of concurrent connections is to move the entire application into kernel space. Recent efforts in this direction have produced dramatic results for Web servers (e.g., TUX [15]). However, this does not obviate the need for efficient event-dispatch mechanisms. In fact, it is our contention that due to security and robustness concerns, many server sites are likely to prefer running Internet servers in user space, provided that they can achieve performance that is comparable to a kernel space solution. Efficient event dispatch mechanisms are also essential for those applications that may be important for some sites (e.g., ftp), but perhaps not quite important enough to warrant the effort of developing an OS-specific kernel solution.

In this paper, we look at the different Linux event-dispatch mechanisms used by servers for doing network I/O. We try to identify the potential bottlenecks in each case, with an emphasis on the scalability of each mechanism and its performance under high load. We use two metrics to determine the efficiency of each mechanism, namely, the event-dispatch overhead and the dispatch throughput. The mechanisms we study in particular are the `select()` system call, `/dev/poll` interface and POSIX.4 Real Time signals (RT signals), each of which is described in more detail in the following sections. Our studies show that RT signals are an efficient and scalable mechanism for handling high loads, but have some potential limitations. We propose an enhancement to the kernel implementation of RT signals that overcomes some of these drawbacks, and allows for robust performance even under high load. We also measure

the performance of a variant of the `select()` based server which amortizes the cost of each `select()` call, and show that it is scalable to a large extent in terms of the server throughput.

The rest of the paper is organized as follows. In Section 2, we describe the primary event-dispatch mechanisms supported by the Linux kernel, and discuss some of the previous work in this regard. In Section 3, we compare some of these mechanisms for their dispatch overhead. We discuss RT signals in more detail, identifying their limitations and propose an enhancement to the default implementation of RT signals in the Linux kernel. In Section 4, we present a comparative study of some of the mechanisms from the perspective of throughput achieved under high loads. Finally, we present our conclusions in Section 5.

2 Event-Dispatch Mechanisms

In this section, we first discuss the two main schemes employed by servers for handling multiple connections. Next, we look at the various event-dispatch mechanisms supported by the Linux kernel that can be employed by Web servers for doing network I/O. We follow this up with a discussion of previous work that has focussed on the scalability of some of these mechanisms, including some other mechanisms that have been proposed to overcome some of the drawbacks of existing mechanisms.

2.1 Handling Multiple Connections

There are two main methodologies that could be adopted by servers for doing network I/O on multiple concurrent connections.

- *Thread-based:* One way to handle multiple connections is to have a master thread accepting new connections, which hands off the work for each connection to a separate service thread. Each of these service threads is then responsible for doing the network I/O corresponding to its connection. These service threads can be spawned in two ways:
  - *On-demand:* Each service thread is forked whenever a new connection is accepted, and it then handles the requests for the connection. This can lead to large forking overhead under high load when there are large number of new connections being established.
  - *Pre-forked:* The server could have a pool of pre-forked service threads. Whenever the master thread receives a new connection, it can hand over the connection to one of the threads from the pool. This method prevents the forking overhead, but may require high memory usage even under low loads.
- *Event-based:* In an event-based application, a single thread of execution uses non-blocking I/O to multiplex its service across multiple connections. The OS uses some form of event notification to inform the application when one or more connections require service. For this to work, the application has to inform the OS of the set of connections (or,

more accurately, the set of file-descriptors) in which it is interested (*interest set*). The OS then watches over the interest set and notifies the server whenever there's activity on any of these connections by dispatching an event to the application. Depending on the exact event-dispatch mechanism in use, the OS could group multiple notifications together or send individual notifications. On receiving the events, the server thread can then handle the I/O on the relevant connections.

In general, thread-per-connection servers have the drawback of large forking and context-switching overhead. In addition, the memory usage due to threads' individual stack space can become huge for handling large number of concurrent connections. The problem is even more pronounced if the operating system does not support kernel-level threads, and the application has to use processes or user-level threads. It has been shown that thread-based servers do not scale well at high loads [7]. Hence, many servers are structured as event-based applications, whose performance is determined by the efficiency of event notification mechanisms they employ. Pure event-based servers do not scale to multiprocessor machines, and hence, on SMP machines, hybrid schemes need to be employed, where we have a multi-threaded server with each thread using event-handling as a mechanism for servicing concurrent connections. Even with a hybrid server, the performance of event-based mechanisms is an important issue. Since efficient event dispatching is at the core of both event-based and hybrid servers, we will focus on the former here.

2.2 Linux Kernel Mechanisms

As described above, event-based servers employ event-dispatch mechanisms provided by the underlying operating system to perform network I/O. In this section, we describe the mechanisms supported by the Linux kernel for event notification to such applications. Following are the mechanisms supported by the Linux kernel.

- `select()` *system call:* `select()` allows a single thread or process to multiplex its time between a number of concurrently open connections. The server provides a set of file-descriptors to the `select()` call in the form of an `fdset`, which describes the interest set of the server. The call returns the set of file-descriptors which are ready to be serviced (for read/write, etc.). This ready set is also returned by the kernel in the form of an `fdset`.

  The main attributes of the `select()` based approach are:

  - The application has to specify the interest set repeatedly to the kernel.
  - The interest set specification could be sparse depending on the descriptors in the set, and could lead to excess user-kernel space copying. The same applies when returning the ready set.
  - The kernel has to do a potentially expensive scan of the interest set to identify the ready file descriptors.

```
// Accept a new connection
int sd = accept(...);

// Associate an RT signal
// with the new socket
fcntl(sd, F_SETOWN, getpid());
fcntl(sd, F_SETSIG, SIGRTMIN);

// Make the socket non-
// blocking and asynchronous
fcntl(sd, F_SETFL, O_NONBLOCK|O_ASYNC);
```

Figure 1: Associating a new connection with an RT signal

  - If the kernel wakes up multiple threads interested in the same file descriptor, there could be a *thundering herd* problem, as multiple threads could vie for I/O on the same descriptor. This, however, is not a problem with Linux 2.4.0 kernel, as it supports single thread wake-up.

- `poll()` *system call:* `poll()` is a system call identical to `select()` in its functionality, but uses a slightly different interface. Instead of using an `fdset` to describe the interest set, the server uses a list of `pollfd` structures. The kernel then returns the set of ready descriptors also as a list of `pollfd` structures. In general, `poll()` has a smaller overhead than `select()` if the interest set or ready set is sparse and a larger overhead if it is dense. Other than that, `poll()` has the same problems as `select()`.

- `POSIX.4` *Real Time Signals*: POSIX.4 Real Time signals (RT signals) are a class of signals supported by the Linux kernel which overcome some of the limitations of traditional UNIX signals. First of all, RT signals can be queued to a process by the kernel, instead of setting bits in a signal mask as for the traditional UNIX signals. This allows multiple signals of the same type to be delivered to a process. In addition, each signal carries a *siginfo* payload which provides the process with the context in which the signal was raised.

  A server application can employ RT signals as an event notification mechanism in the following manner. As shown in figure 1, the server application can associate an RT signal with the socket descriptors corresponding to client connections using a series of `fcntl()` system calls. This enables the kernel to enqueue signals for events like connections becoming readable/writable, new connection arrivals, connection closures, etc. Figure 2 illustrates how the application can use these signal notifications from the kernel to perform network I/O. The application can block the RT signal associated with these events (`SIGRTMIN` in figure 2) and use `sigwaitinfo()` system call to synchronously dequeue the signals at its convenience. Using `sigwaitinfo()` obviates the need for asynchronous signal delivery and saves the overhead of invoking a signal handler. Once it fetches a signal, the *siginfo* signal payload enables the application to identify the socket descriptor for which the signal was queued. The application can then perform the appropriate action on the socket.

```
sigset_t signals;
siginfo_t siginfo;
int signum, sd;

// Block the RT signal
sigemptyset(&signals);
sigaddset(&signals, SIGRTMIN);
sigprocmask(SIG_BLOCK, &signals, 0);

while (1) {
    // Dequeue a signal from the signal queue
    signum = sigwaitinfo(&signals, &siginfo);

    // Check if the signal is an RT signal
    if (signum == SIGRTMIN) {
        // Identify the socket associated with the signal
        sd = siginfo.si_fd;
        handle(sd);
    }
}
```

Figure 2: Using RT signals for doing network I/O

One problem with RT signals is that the signal queue is finite, and hence, once the signal queue overflows, a server using RT signals has to fall back on a different dispatch mechanism (such as `select()` or `poll()`). Also, `sigwaitinfo()` allows the application to dequeue only one signal at a time. We'll talk more about these problems in the next section.

2.3 Previous Work

Banga et al. [1] have studied the limitations of a `select()` based server on DEC UNIX, and shown the problems with its scalability, some of which we have discussed above. They have proposed a new API in [2], which allows an application to specify its interest set incrementally to the kernel and supports event notifications on descriptors instead of state notifications (as in the case of `select()` and `poll()`). The system calls proposed as part of this API are `declare_interest()`, which allows an application to declare its interest in a particular descriptor, and `get_next_event()`, which is used to get the next pending event(s) from the kernel.

Another event-dispatch mechanism is the `/dev/poll` interface, which is supported by the Solaris 8 kernel [12]. This interface is an optimization for the `poll()` system call. Recently, Provos et al. [10] have implemented the `/dev/poll` interface in the Linux kernel. This interface allows the interest set to be described incrementally by writing to the `/dev/poll` device. The polling is done by using an `ioctl()` call, which returns a list of `pollfd` structures corresponding to the set of ready descriptors.

RT signals have been used for network I/O in *phhttpd* [4] Web server. Provos et al. have discussed its implementation and some of its shortcomings and also proposed a new system call `sigtimedwait4()` for dequeuing multiple signals from the signal queue [11].

3 Dispatch Overhead

In this section, we look at the first scalability parameter for event-dispatch mechanisms, namely the overhead involved in handling requests as a function of the number of concurrent requests. This parameter becomes important in the context of large number of idle or slow connections, irrespective of the actual active load on the server. In what follows, we first present an experimental study of some of the Linux dispatch mechanisms, and then discuss some of the insights from this study. We follow this up with a detailed discussion of RT signal behavior, including their limitations. We then propose an enhancement to the implementation of RT signals in the Linux kernel to overcome some of these limitations.

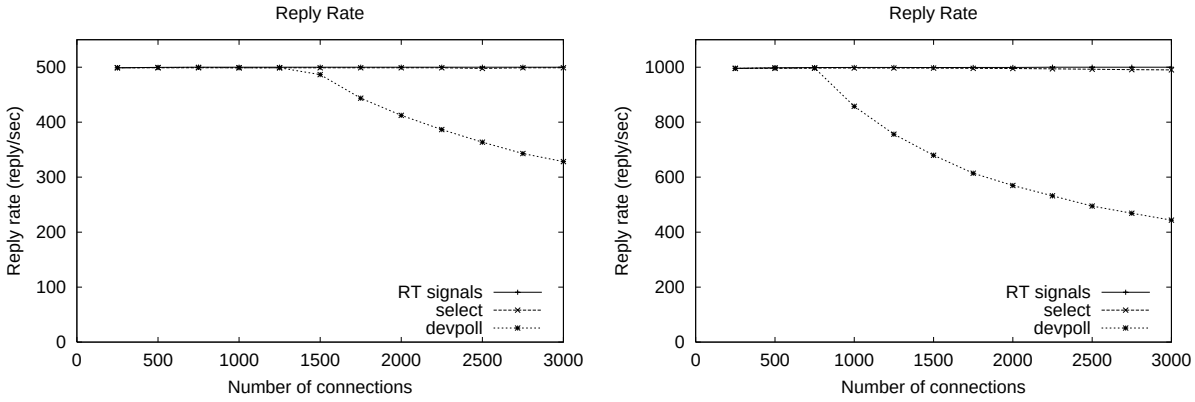
3.1 Comparative Study

In what follows, we present the results of our comparative study of some of the kernel mechanisms discussed above. The main goal of this study was to look at the behavior of Web servers under high load in terms of their CPU usage as the number of concurrent connections (most of them idle) increases.

3.1.1 Experimental Testbed

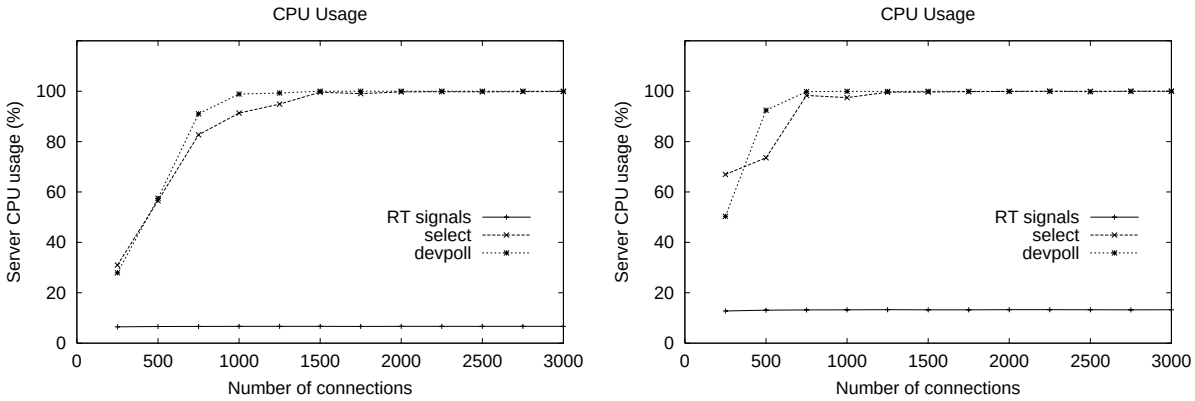
To conduct the experimental study, we implemented a set of *micro Web servers (μservers)*, each employing a different event-dispatch mechanism. Most of the request handling and administrative code in these μservers is identical to avoid differences in performance arising due to other factors. Apart from trying to ensure that the different versions are as similar as possible, using them instead of widely-used, full-fledged Web servers allows us to focus on the performance impact of the dispatch mechanisms by reducing all other overheads to the absolute minimum. Thus, the μservers do very simple HTTP protocol processing, and the various μservers differ only in their use of the event-dispatch mechanism. Specifically, we compared μservers employing `select()`, `/dev/poll` and RT signals as their event-dispatch mechanisms.

Each of these μservers was run on a 400 MHz Pentium-III based dual-processor HP NetServer LPr machine running Linux 2.4.0-test7 in uniprocessor mode. The client load was generated by running *httperf* [8] on ten B180 PA-RISC machines running HP-UX 10.20. The clients and the server were connected via a 100 Mbps Fast Ethernet switch. To simulate large number of concurrent and idle connections, each httperf was used to establish a set of persistent connections, each of which generated periodic requests to the μserver. The effect was that at all times, some of the connections were active while the rest were idle, and these active and idle



(a) 500 req/s (b) 1000 req/s

Figure 3: Reply rate with varying number of concurrent connections



(a) 500 req/s (b) 1000 req/s

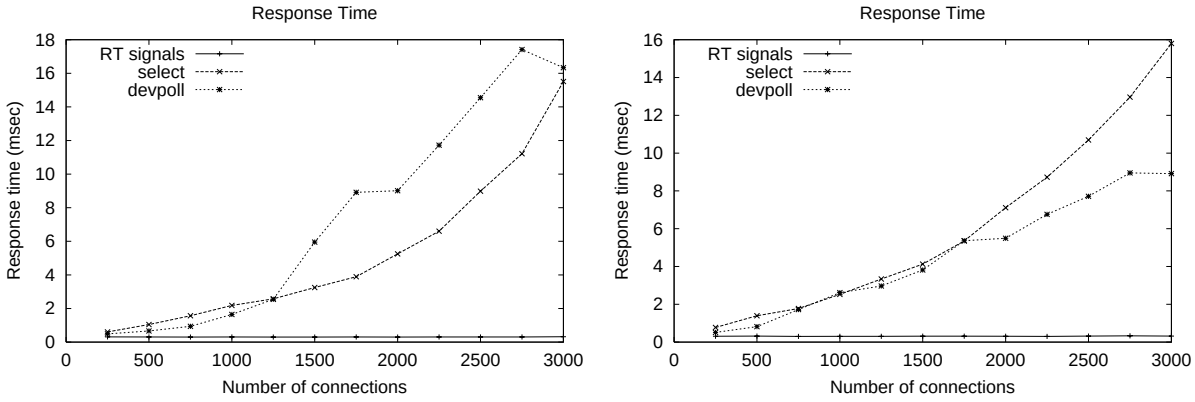
Figure 4: CPU usage with varying number of concurrent connections

connection sets kept changing with time. The server's reply size was 92 bytes. In each experiment, the total request rate was kept constant, while the number of concurrent connections was varied to see the effect of large number of idle connections on server performance.

To measure the CPU usage of the μserver, we inserted an `idle_counter` in the kernel running the μserver. This `idle_counter` counted the idle cycles on the CPU. We computed the CPU load imposed by the μserver by comparing the idle cycles with the μserver running on the system to those for an unloaded system.

3.1.2 Experimental Results

As part of our comparative study, we ran experiments to measure the performance of three μservers based on `select()`, `/dev/poll` and RT signals respectively. In each experiment,



(a) 500 req/s (b) 1000 req/s

Figure 5: Response time with varying number of concurrent connections

the clients were used to generate a fixed request rate, and the number of concurrent connections was increased from 250 to 3000. Figure 3 shows the reply rates achieved by the servers for request rates of 500 req/s and 1000 req/s respectively. The reply rate matches the request rate for the RT signal and `select()` based servers at all points. On the other hand, the reply rate starts dropping off for the `/dev/poll` based server after a point. This is because the server becomes overloaded and starts dropping connections beyond a certain load. The reason why the `/dev/poll` server performs so poorly under overload might be due to a bug in the `/dev/poll` implementation.

The more interesting figures are figures 4 and 5, which show the CPU usage and the average response time respectively for each of the μservers, as the number of concurrent connections is increased. As can be seen from figure 4, the CPU usage for both `select()` and `/dev/poll` increases with the number of concurrent connections and they become saturated after a certain point. On the other hand, the CPU usage for RT signals is insensitive to the number of idle connections. The RT signal based server's CPU usage is about 6.67% on average for the 500 req/s case, while it is about 13.25% for the 1000 req/s case. Thus, the CPU overhead of RT signals seems to be dependent only on the request rate. Also, the RT signal CPU usage is dramatically lower than either `select()` or `/dev/poll` based servers. A similar behavior is seen for the response time in figure 5. Once again, the response time increases for both the `select()` and `/dev/poll` based servers with the number of connections. On the other hand, the RT signal based server shows a very small response time for each of the request rates (about 0.3 ms in each case). Further, this response time is independent of the number of concurrent connections.

Thus, the results in this section show that RT signals have very small dispatch overhead and also that this overhead does not depend on the number of concurrent or idle connections being handled by the server. Rather, it is determined only by the active work being done by the server.

3.2 RT Signals: Reasons for Efficiency

From our comparative study, we observe that RT signals have a relatively low overhead compared to `select()` and `/dev/poll` event-dispatch mechanisms. Further, this overhead seems to be independent of the number of idle connections, and depends only on the active request rate. In other words, RT signals show essentially ideal behavior. In this section, we discuss the reasons for the better performance of RT signals in more detail.

RT signals are more efficient due to the following reasons:

- First, the server only needs to specify its interest set to the kernel incrementally. This is because the server application associates an RT signal with each socket file descriptor at the time of its creation (just after the `accept()` system call). From this point onwards, the kernel automatically generates signals corresponding to events on the descriptor, and thus obviates the need for the application to specify its interest in the descriptor again and again (as is the case with `select()` system call). This functionality is similar to the `declare_interest()` API proposed in [2].
- Unlike `select()`, `poll()` and `/dev/poll`, in the case of RT signals, the kernel does not know about the interest set explicitly. Rather, whenever there's an event on one of the descriptors, the kernel enqueues a signal corresponding to the event without having to worry about the interest set. Thus, the interest set is totally transparent to the kernel and this gets rid of the overhead of scanning each descriptor in the interest set for activity on every polling request from the application.
- Based on the `fd` field in the signal payload, the application can identify the active descriptor immediately without having to potentially check each descriptor in the interest set (as in the case of `select()`).
- By blocking the relevant RT signal and using `sigwaitinfo()` for dequeuing signals from the signal queue, the overhead of calling a signal handler is avoided.

3.3 Limitations of RT signals

In spite of their efficiency, RT signals, as currently implemented in Linux, have some potential limitations. These limitations arise from the fact that the signal queue is a limited resource. Since each event results in a signal being appended to the signal queue, a few active connections could dominate the signal queue usage or even trigger an overflow. The former could result in unfair service and the latter could cause a deadlock-like situation in which the server can no longer make any progress.

To understand how a signal queue overflow can lead to a deadlock situation, note that once the queue is full, no further signals can be enqueued and hence all future events are dropped. Of course, eventually the server would drain the queue and new events would start to come in again. However, those events that got dropped are lost forever. If one of those events happened to indicate, for example, that the listen queue has pending connections, the server

may never realize that it ought to call `accept()` to service those connections. Similarly, if an event got dropped that indicated that a particular connection is now readable, the server may never realize that it should call `read()` on that connection. Over time, the more events are dropped, the more likely it becomes that either some connections end up in a suspended state or that the listening socket is no longer serviced. In either case, throughput will suffer and eventually drop to zero.

To avoid this kind of deadlock, the Linux kernel sends a `SIGIO` signal to the application when the signal queue overflows. At this point, the application can recover from the overflow by falling back onto some other event dispatch mechanism. For example, the application could use `select()` or `poll()` to detect any events that may have been dropped from the signal queue. Unfortunately, using a fallback mechanism comes with its own set of problems. Specifically, there are two issues:

- First, having to handle signal queue overflows by switching onto another mechanism makes the application complex. It may require translating the interest set from the (implicit) form used by the RT signal mechanism into the explicit form used by the other mechanism. Furthermore, the application has to receive and service the kernel notifications in a different manner. Also, this transition needs to be done very carefully, as losing even a single event could potentially create the deadlock-like situation mentioned above.
- Second, switching over to a non-scalable mechanism also has the potential to make the application sluggish. Since the application is already under overload (which led to the signal queue overflow in the first place), using a high-overhead mechanism for recovery could overload the server even further, potentially sending it into a tailspin.

Another drawback with RT signals is that each call to `sigwaitinfo()` dequeues exactly one signal from the queue. It cannot return multiple events simultaneously, which might be a problem under high load.

Thus, using RT signals as implemented in the kernel has some potential drawbacks even if they are used in conjunction with another mechanism.

3.4 Signal-per-fd: RT Signal Enhancement

As discussed above, having to handle a signal queue overflow could be potentially costly as well as complex for an application. It would be desirable, therefore, if signal queue overflows could be avoided altogether. To understand why signal queue overflows are happening in the first place, note that there's a potential of multiple events being generated for each connection, and hence multiple signals being enqueued for each descriptor. But, most of the time, the application does not need to receive multiple events for the same descriptor. This is because even when an application picks up a signal corresponding to an event, it still needs to check the status of the descriptor for its current state, as the signal might have been enqueued much before the application picks it up. In the meantime, it is possible that there might

have been other events and the status of the descriptor might have changed. For instance, the application might pick up a signal corresponding to a read event on a descriptor *after* the descriptor was closed, so that the application would have to decide what to do with the event in this case. Thus, it might be more efficient and useful if the kernel could coalesce multiple events and present them as a single notification to the application. The application could then check the status of the descriptor and figure out what needs to be done accordingly.

We propose an enhancement to achieve this coalescing, which we call *signal-per-fd*. The basic idea here is to enqueue a single signal for each descriptor. Thus, whenever there's a new event on a connection, the kernel first checks if there's already a signal enqueued for the corresponding file descriptor, and if so, it does not add a new signal to the queue. A new signal is added for a descriptor only if it does not already have an enqueued signal.

To efficiently check for the existence of a signal corresponding to a descriptor, we maintain a bitmap per process. In this bitmap, each bit corresponds to a file-descriptor and the bit is set whenever there is an enqueued signal for the corresponding descriptor. Note that checking the bit corresponding to a descriptor obviates the need to scan the signal queue for a signal corresponding to the descriptor, and hence, this check can be done in constant time. This bit is set whenever the kernel enqueues a new signal for the descriptor and it is cleared whenever the application dequeues the signal.

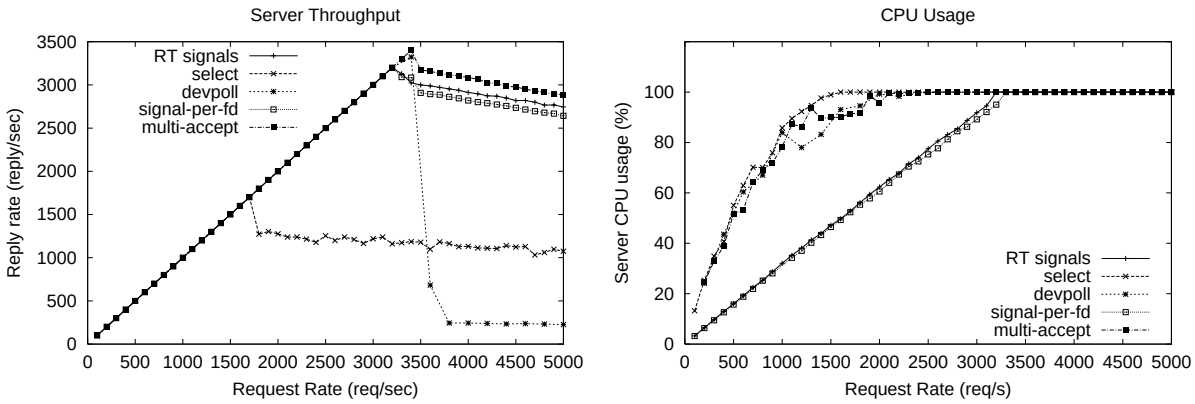
By ensuring that one signal is delivered to the application for each descriptor, the kernel coalesces multiple events for a connection into a single notification, and the application then checks the status of the corresponding descriptor for the action to be taken. Thus, if the size of the signal queue (and hence the bitmap) is as large as the file descriptor set size, we can ensure that there would never be a signal queue overflow.

This enhancement has the following advantages:

- Signal-per-fd reduces the complexity of the application by obviating the need to fall back on an alternative mechanism to recover from signal queue overflows. This means that the application does not have to re-initialize the state information for the interest set, etc. that may be required by the second mechanism. In addition, the application does not have to deal with the complexity associated with ensuring that no event is lost on signal queue overflow. Finally, the server would not have to pay the penalty of potentially costly dispatch mechanisms.

- Signal-per-fd also ensures fair allocation of the signal queue resource. It prevents over-loaded and misbehaving connections from monopolizing the signal queue, and thus achieves a solution for the proper resource management of the signal queue.

- By coalescing multiple events into a single notification, this mechanism prevents the kernel from providing too fine-grained event notifications to the application, especially as the application might not pick up the notifications immediately after the events. This enhancement thus notifies the application that there *were* events on a descriptor, instead of *how many* events there were. The latter information is often useless to the application as it has to anyway figure out *what* the events were and what the status of the descriptor is.



(a) Reply rate (b) CPU usage

Figure 6: Server performance with 252 idle connections

On the whole, signal-per-fd is a simple enhancement to the implementation of RT signals that can overcome some of their limitations in the context of using them as an event-dispatch mechanism for doing network I/O.

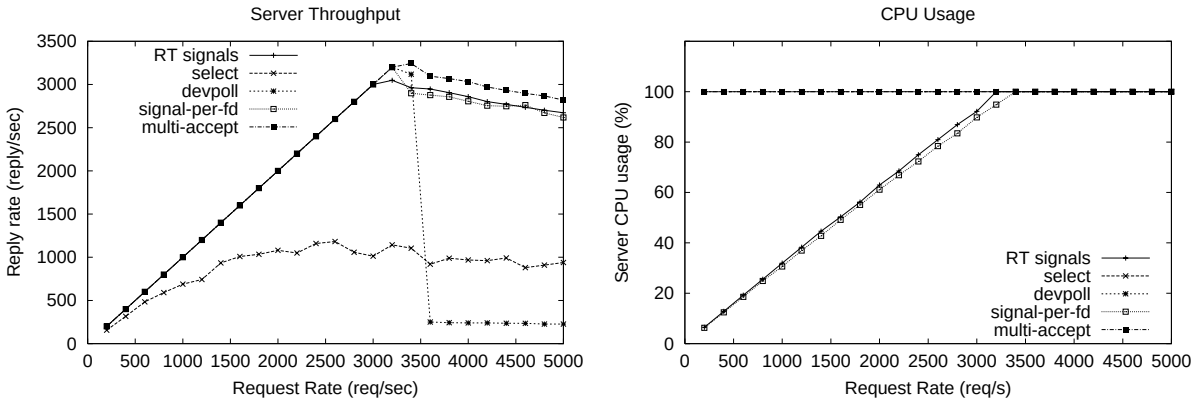
4 Dispatch Throughput

In this section, we look at another parameter associated with the efficiency of event-dispatch mechanisms, namely, the throughput that can be achieved as a function of the load on the server. This metric is orthogonal to the overhead discussed in the previous section, as this refers to the active load on a server, which reflects the actual amount of useful work being performed by the server. In what follows, we first provide a comparative experimental study of some of the Linux dispatch mechanisms, including the *signal-per-fd* optimization proposed in the previous section. In addition, we also look at the throughput achieved by a `select()` based server with a minor modification which allows the server to do multiple `accept()`s each time the listening socket becomes ready. Then, we discuss the results of this study and provide some insights into the behavior of the various mechanisms.

4.1 Experimental Study

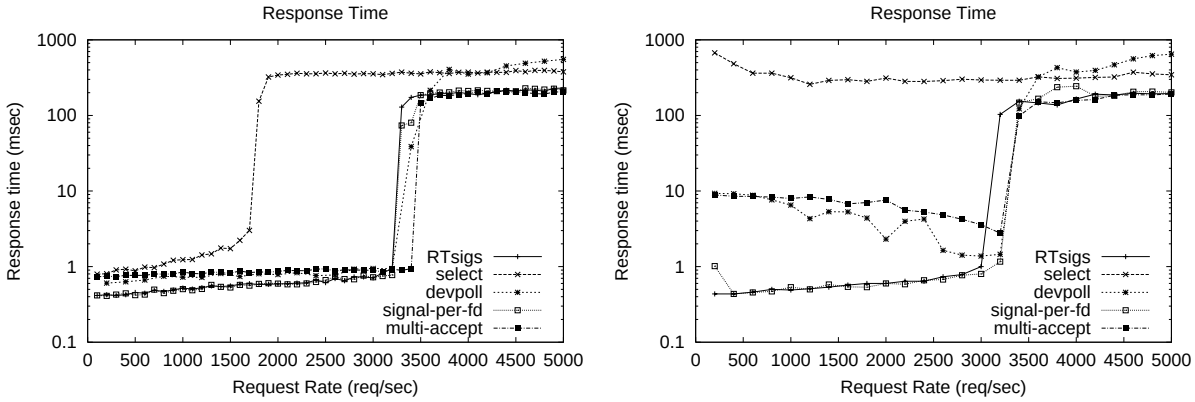
Here, we experimentally evaluate the throughput achieved by various event-dispatch mechanisms under high load. Our experimental setup is the same as that used in Section 3.1 for comparative study of `select()`, `/dev/poll` and RT signals. In this study, we evaluate two new mechanisms/enhancements as well:

(i) The signal-per-fd enhancement to RT signals. We have implemented this enhancement in the Linux 2.4.0-test7 kernel, and we ran the RT signal based μserver on the modified kernel for measuring the effect of signal-per-fd.



(a) Reply rate (b) CPU usage

Figure 7: Server performance with 6000 idle connections



(a) 252 idle connections (b) 6000 idle connections

Figure 8: Response time with increasing load

(ii) A `select()` based μserver which does multiple `accept()`s each time the listening socket becomes ready, as opposed to the standard `select()` based μserver which does only one `accept()` for such a case. We'll refer to this modification as *multi-accept* `select`. The idea here is to enable the server to identify more connections and perform more useful work per `select()` call.

We consider these mechanisms here, but not in the dispatch overhead study, because signal-per-fd would have similar behavior as the standard RT signal implementation unless the server is under overload. Similarly, multi-accept `select` would have overhead similar to the standard `select()` based server in terms of handling concurrent idle connections, and would differ only when the active load on the servers is varied.

In order to measure the throughput of these various mechanisms under varying loads, we used a set of idle connections along with a set of *httperf* clients generating requests at high rates. In these experiments, we kept the number of idle connections fixed for each experiment and varied the request rate. We used a very small reply size (1 byte excluding the header information, etc.) to see how much load the servers could sustain in terms of the number of requests.

Figure 6 shows the performance of the servers with 252 idle connections, while figure 7 plots the same information for 6000 idle connections. As can be seen from figures 6(a) and 7(a), the throughput with `select()` plateaus out much before it does for the RT signals (both the default and the signal-per-fd implementations). The fall in reply rate of `/dev/poll` is much more dramatic and again, it seems to perform very poorly under overload. The interesting observation is that multi-accept `select` is able to sustain a high throughput, similar to the RT signals, and even manages to achieve a slightly higher peak throughput in each case. Figures 6(b) and 7(b) show the CPU usage for the μservers. Again, as can be seen from these figures, the CPU usage for RT signals is much less than that for `select()`, multi-accept `select` and `/dev/poll` in all cases, and RT signals reach saturation at a much higher load. In fact, for 6000 idle connections (figure 7(b)), CPU usage is 100% for the `select()`, multi-accept `select` and `/dev/poll` based μservers right from the beginning, which can be attributed to their high overhead in handling large number of concurrent connections. On the other hand, the CPU overhead for the RT signals based server (for both the default and signal-per-fd cases) increases linearly with the load in either case. An interesting point to be noted from these figures is that the server with the default RT signal implementation reaches saturation at a slightly smaller load than signal-per-fd, and this is more pronounced for the 6000 idle connections. We will discuss this point in more detail below.

Figures 8(a) and (b) plot the average response times of the various servers with increasing load for 252 and 6000 idle connections respectively. Figure 8(a) shows that `select()` reaches overload at a relatively low load, while the other mechanisms get overloaded at much higher loads. In figure 8(b), `select()` shows high response times for all loads and is thus overloaded for all the points in the graph. These plots complement figures 6(a) and 7(a), which show the throughput for these cases. The figures further show that the `/dev/poll` server achieves small response times at low loads, but under overload, it offers much higher response times compared to the other mechanisms. Thus, its overload behavior is again seen to be very poor. The interesting point in figure 8(a) is that multi-accept `select` is able to

Request Rate (req/s)	252 idle connections		6000 idle connections	
	No. of `sigwaitinfos`	No. of `SIGIOs`	No. of `sigwaitinfos`	No. of `SIGIOs`
2800	504728	0	504474	0
3000	540576	0	540792	0
3200	10538	1526	19	19
3400	40	40	16	16
3600	40	40	14	14
3800	39	39	13	13
4000	39	39	13	13

Table 1: Signal queue overflows under high loads

provide a low response time upto very high loads. Figure 8(b) shows an even more interesting behavior of multi-accept `select` — its response time actually *decreases* with increasing load until it hits overload. This behavior clearly shows the load amortization occuring for multi-accept `select`, so that more useful work being extracted for the same `select()` call overhead translates to lower *average* response times. Finally, the two RT signal implementations have the lowest response times until they get overloaded, which is expected as they have the lowest overhead. Once again, these graphs show that the default RT signal based server reaches overload slightly earlier than the signal-per-fd server.

Next, we will try to understand these results, and in particular, we will focus on the behavior of multi-accept `select` and the two implementations of RT signals.

4.2 Discussion

From the results of our comparative study, we get the following insights into the behavior of the various mechanisms:

- Using multi-accept `select` increases the throughput of the `select()` based server substantially. This is because `select()` is basically a state notification mechanism. Thus, when it returns the listening socket as ready, it means there are new connections queued up on the listen queue. Using multiple `accept()` calls at this point drains the listen queue without having to call `select()` multiple times. This helps prevent the high cost of using multiple `select()` calls for identifying new connections. Once new connections are efficiently identified and added to the interest set, under high load, `select()` would have large number of active connections to report each time it is called. Thus, its cost would be amortized as the server could perform more useful work per `select()` call. This has to do with the fact that the ready set returned by `select()` would be dense and hence, the scanning cost of `select()` is utilized better. Also, the more useful work the server does on each call to `select()`, the less often it needs to be called. Hence, the server is able to identify more connections and extract more useful work, and thus achieves a higher throughput. Note that the overhead is still high — only the overhead is being *better utilized* now.

The high throughput achieved by the multi-accept `select()` server is contrary to conventional wisdom, according to which `select()` based servers should perform poorly under high loads in terms of their throughput as well. While this is true for a simple `select()` based server, our results show that implementing the server more carefully can help us achieve better performance.

- The behavior of the servers running on the default RT signal implementation and the signal-per-fd implementation are very similar until saturation. This is understandable as there are very few signal queue overflows under low loads, and hence, the two schemes work essentially the same. To verify that this is indeed the case, in table 1, we have tabulated the number of `SIGIO`s and the number of `sigwaitinfo()` calls for the default RT signal based server, at some of the loads for 252 and 6000 idle connections respectively. The number of `sigwaitinfo()`s denotes the number of times the server dequeued a signal from the signal queue, and the number of `SIGIO`s represents the number of signal queue overflows. As can be seen from the table, under low loads, there are no `SIGIO`s and hence, no signal queue overflows. On the other hand, at high loads, all the `sigwaitinfo()` calls result in `SIGIO`s. This indicates that the signal queue is overflowing all the time, and hence, the server has to fall back on an alternative mechanism to perform its network I/O under high loads. The fallback mechanism used in our server was multi-accept `select`[1]. Hence, under high loads, the default RT signal server behavior is identical to that of the multi-accept `select` server, as was seen from the throughput and the response time plots.

- As noted earlier, the signal-per-fd server reached overload at slightly higher load compared to the default RT signal server. In particular, the default RT signal based server saturated at about 3200 req/s, which corresponds to the high number of `sigwaitinfo()`s resulting in `SIGIO`s at this load, as can be seen from table 1. Thus, preventing signal queue overflows seems to make the server sustain slightly higher loads before getting saturated. Once the signal-per-fd server becomes saturated, its throughput is bounded by the amount of useful work it can amortize over each signal notification, even though it does not suffer from signal queue overflows. Recall that similar to `select()`, signal-per-fd is also a state-notification mechanism — hence the server can extract more work per signal compared to the default event-notification mechanism. Thus, its throughput is comparable to that of multi-accept `select` under overload, even though its peak throughput is slightly smaller, as `sigwaitinfo()` still returns only one signal per call.

To summarize, we find that RT signals are an efficient mechanism in terms of overhead, and under saturation, their throughput is determined by the fallback mechanism being used to handle signal queue overflows. We find that `select()` system call can give high throughput if we use multiple `accept()`s to identify more new connections per `select()` call. Finally, signal-per-fd has a behavior almost identical to that of the default RT signal implementation in terms of overhead and throughput, but it is able to sustain slightly higher load before

[1]We cannot simply use `select()` with single `accept()` in this situation because, to prevent any potential deadlocks, we have to ensure that no event is lost, and hence, we need to clear the listen queue completely.

becoming overloaded. Further, it helps reduce the complexity of the server to a large extent. This is because we do not have to worry about using alternative event-dispatch mechanisms, and state maintenance also becomes much easier.

5 Conclusion

In this paper, we first discussed some of the common event-dispatch mechanisms employed by Internet servers. We focussed on the mechanisms available in the Linux kernel, and measured their performance in terms of the overhead and throughput of a minimal Web server. Our comparative studies showed that RT signals are a highly efficient mechanism in terms of their dispatch overhead and also provide good throughput compared to mechanisms like `select()` and `/dev/poll`. In particular, the overhead of RT signals is independent of the number of connections being handled by the server, and depends only on the active I/O being performed by it. But, an RT signal based server can suffer from signal queue overflows. Handling such overflows leads to complexity in the server implementation and also potential performance penalties under high loads. To overcome these drawbacks, we proposed a scheme called *signal-per-fd*, which is an enhancement to the default RT signal implementation in the Linux kernel. This enhancement was shown to significantly reduce the complexity of a server implementation, increasing its robustness under high load, and also potentially increasing its throughput. Overall, we conclude that RT signals are a highly scalable event-dispatch mechanism and servers based on these signals can also be substantially simplified when coupled with the signal-per-fd enhancement.

Another interesting result of our study was the performance of `select()` based servers under high loads. According to conventional wisdom, `select()` based servers have high overhead and thus, perform very poorly under high loads in terms of the server throughput as well. Our experiments with the *multi-accept* variant of a `select()` based server show that though `select()` does have high dispatch overhead, this overhead can be amortized better by performing more useful work per `select()` call, resulting in a high throughput even under heavy load conditions. Thus, we conclude that even a `select()` based server can be made to scale substantially if its overhead is better utilized to perform more useful work.

Acknowledgements

We would like to thank Martin Arlitt for providing us with large number of client machines and helping us set up the test-bed for our experiments.

References

[1] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performace for Internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[2] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.

[3] Gordon Bell and Jim Gemmell. On-ramp prospects for the Information Superhighway Dream. *Communications of the ACM*, 39(7):55–61, July 1996.

[4] Z. Brown. phhttpd, November 1999.
http://www.zabbo.net/phhttpd.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, January 1997.

[6] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly, 1995.

[7] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks. In *Proceedings of the Second IEEE Global Internet Conference*, November 1997.

[8] David Mosberger and Tai Jin. httperf – A Tool for Measuring Web Server Performance. In *Proceedings of the SIGMETRICS Workshop on Internet Server Performance*, June 1998.

[9] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.

[10] Niels Provos and Chuck Lever. Scalable Network I/O in Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2000.

[11] Niels Provos, Chuck Lever, and Stephen Tweedie. Analyzing the Overload Behavior of a Simple Web Server. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, October 2000.

[12] *Solaris 8 man pages for* `poll(7d)`.
http://docs.sun.com:80/ab2/coll.40.6/REFMAN7/@Ab2PageView/55123?Ab2Lang=C&Ab2Enc=iso-8859-1.

[13] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[14] thttpd – tiny/turbo/throttling HTTP server.
http://www.acme.com/software/thttpd.

[15] Answers from Planet TUX: Ingo Molnar Responds (*interview*).
http://slashdot.org/interviews/00/07/20/1440204.shtml.

[16] Zeus Web Server.
http://www.zeustech.net/products/ws.