A Scalable Event Dispatching Library for Linux Network Servers

Hao-Ran Liu and Tien-Fu Chen

Department of Computer Science
National Chung Cheng University
Chiayi, Taiwan 621, ROC

Abstract

Network servers often want to process network events asynchronously. They use `select()` or `poll()` to get events on file descriptors. However, previous research has shown that these two system calls scale poorly when the number of open connections are increased. In this paper, we compare several event dispatching mechanisms available under Linux, and present our user-mode solution that takes advantage of temporal locality among events while polling. We show that a memory-based web server with this change can have about 20%-30% performance improvement.

Keywords: Scalable, Event Dispatching, Network Server

1 Introduction

Scalability of network servers is important today. As the Internet continues to grow in popularity and size, network servers without good scalability will result in performance drop or live-lock[8].

Traditional web servers like Apache[1] serve every connection with one dedicated process. This approach simplifies code complexity, but blocks threads calling `read()` or `write()` inside kernel. Servers rely on OS kernel to switch context between processes to provide concurrent services to all

```
struct pollfd ev_array[MAX_FD];
int num_event, ev_num, i;

while (TRUE) {
    num_event = poll(ev_array, ev_num, 0);
    if (num_event == 0) {
        do_sleep(msec);
        continue;
    }

    for (i = 0; i < num_event; i++) {
        if (ev_array[i].revents & POLLIN)
            read_handler(ev_array[i].fd);
        else if (ev_array[i].revents & POLLOUT)
            write_handler(ev_array[i].fd);
    }
}
```

Figure 1: Using `poll()` to detect network events

connections. When the number of connections increase, number of processes increases and a large portion of CPU time is wasted on context switches.

Another approach is to serve all connections in a single process, called single process event-driven (SPED). Server cannot block itself waiting for any read or write on a file descriptor. Nonblocking I/O is needed. However, calling `read()` or `write()` excessively on all file descriptors to see if they are ready for read or write is inefficient. We must use system call like `poll()` to inquire kernel which file descriptor is ready and only read or write on ready one. The goal of this approach is to reduce context switching and synchronization overhead. Squid[12], a famous web proxy cache, is based on SPED architecture. Figure 1 is a simplified code demonstrating how `poll()` is used.

Although single process event driven (SPED) architecture is much more efficient than multiple process (MP) architecture. Previous research[2, 10] has shown that `poll()` is not scalable; more than 30% of CPU time is spent on such a system call on a normal squid proxy server. `poll()` performs the amount of work in proportion to the number of file descriptors in event array rather than constant factor.

On a regular web server, most connections are idle, because users must think before they click on next URL and packets may be delivered across a network that suffers from traffic congestion. `poll()` spends most of time on polling useless idle connections. HTTP 1.1 persistent connection allows multiple requests in a single connection. As more and more web clients support HTTP 1.1 protocol, persistent connection will increase connection time and make server polling on idle connections performance worse.

In this paper, our focus is on improving web server performance in user-mode. We compare different event dispatching mechanisms in Linux, and give a summary of them. Based on the observation that most web connections are idle, we present a library managing file descriptors and calls `poll()` with temporal locality in mind. Performance evaluation of the library is done on our memory-based event-threading server. Our studies show that performance of the server can be significantly improved by 30% or more. Programming interface and implementation details of web server and library are discussed. Performance analysis is given based on two metric: event dispatching overhead and dispatching throughput.

The rest of the paper is organized as follows. Section 2 describes various event dispatching mechanisms in Linux. Section 3 shows our polling strategy. Section 4 describes programming interface and how to integrate it into your own code. Section 5 talks about implementation details of the library and important parameters in server code that may influence overall performance. Section 6 evaluates performance of the library and other event dispatching mechanisms. We give conclusion and future work in Section 7.

2 Event Dispatching Mechanisms in Linux

There are many event dispatching mechanisms introduced and being discussed in the literature. Some of them are already incorporated into Linux 2.4 and some are only available in the form of kernel patch. Here we give an overview of these mechanisms and discuss their advantages and

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

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

int poll(struct pollfd *ufds,
         unsigned int nfds;
         int timeout);
```

Figure 2: Prototype of `select()` and `poll()`

disadvantages.

2.1 `poll()` and `select()`

`poll()` and `select()` are state-based event dispatching mechanisms. They report current states of a set of file descriptors specified in arguments. Their implementation are similar. The only difference is their system call interface. Figure 2 shows the interface of the two system calls.

`select()` uses three long bitmaps `fd_set` to represent the set of file descriptors that are interesting for reading, writing and exceptional conditions. `nfds` argument tells the kernel the largest bitmap index actually used. `timeout` specifies the time to wait inside kernel before `select()` returns if there is no event at all for all file descriptors. Application has to set corresponding bit on the three sets for all interesting descriptors before calling `select()`. On return, kernel overwrites these sets with new values, which is a subset of input sets, telling readiness of each interesting descriptors.

`poll()` uses an array of `pollfd` structure for request events and return events of interesting file descriptors. `events` field tells the kernel of interesting events of a file descriptor. On return, `revents` is filled by kernel to indicate readiness of read, write or exception. `nfds` indicates the size of `ufds` array.

When the number of file descriptors is large, `select()` is more suitable because fewer data is copied to and from kernel. However, both `poll()` and `select()` is not scalable when a server is overloaded with a large set of file descriptors. Because kernel must do a linear scan on all interesting descriptors and call device driver's poll callback respectively. In addition, when either `poll()` or `select()` returns, application does another linear scan on return value. Frequently execution of `poll()` or `select()` and the two linear scans make a network server like web server and proxy server scales poorly when it is overloaded with connections.

2.2 POSIX.4 Real Time Signal

POSIX.4 real time signal[4] (*RT signals*) is an extension to traditional UNIX signal. It allows multiple the same signals to be queued by kernel for per process. RT signal is delivered with a `siginfo` payload. Information like sender process ID and sender user ID are carried together with signal delivery.

Linux 2.4 extends POSIX.4 RT signal, allowing delivery of socket readiness via a particular real time signal. Though this feature is Linux specific, it scales pretty well with large set of descriptors. RT signals are event-based event dispatching mechanism. Events are reported via signal queue if there is any file descriptors that become ready. Readiness of descriptors are put into signal queue at the arrival time of network events. Later, when a server calls get-event system call, expensive calls of device driver's poll callback can be eliminated. To employ RT signals in a server application, we call `fcntl()` first to associate a file descriptor with an RT signal. Then, we block the RT signal and use `sigwaitinfo()` or `sigtimedwait()` to dequeue signals synchronously. Figure 3 illustrates how a RT signal is associated with a file descriptor. Notice that asynchronous I/O must be enabled for the occurrence of RT signals. Figure 4 demonstrates how RT signals are used on a single process web server. Signals associated with file descriptors must be blocked to avoid signal handlers being invoked. When `sigwaitinfo()` receives `SIGIO`, it means that the signal queue in the kernel is overflowed and

```
// accept a new connection
int sfd = accept(conn, ...);

// associate an RT signal with a connection
fcntl(sfd, F_SETSIG, SIGRTMIN);
// set the process ID to receive the signal
fcntl(sfd, F_SETOWN, getpid());

// Enable nonblocking and asynchronous I/O
fcntl(sfd, F_SETFL, O_NONBLOCK | O_ASYNC);
```

Figure 3: Associating an RT signal with a connection

some events are discarded. Event loss will make some connections deadlock. To avoid this situation, we need to clean up signal queue and fall-back to traditional `poll()` or `select()` to scan for events on all file descriptors. Notice that the program in Figure 4 always calls `poll()` first, this is because when a event comes after application calls `accept()` but before `fcntl(F_SETSIG, ...)`, the event will be lost. To correct this, server application needs call `poll()` first before RT signals are used.

Chandra and Mosberger introduced *signal-per-fd*[3] to prevent signal queue from being overflow. *signal-per-fd* collapses events of the same descriptor in the signal queue. If signal queue length is equal to maximum number of file descriptors a process can have, an overflow of signal queue will not happen. Although this prevents signal queue from overflow, code related to `poll()` still can't be removed, because RT signals doesn't return initial state of a descriptor when users associate it with an RT signal. Luban's Linux patch for *signal-per-fd* is available at [11].

The disadvantage of RT signals is the complexity of server code comparing to other mechanisms. Only one fetched event per `sigwaitinfo()` call also leads to many switches between user mode and kernel mode. The advantage of RT signals is that it is scalable and users can associate some descriptors with one signal, while some with another. This divides descriptors into several "interesting sets" and allows server to process them differently.

```
struct pollfd ev_array[MAX_FD];
struct timespec wtime = {0, 0};
sigset_t sset; siginfo_t sinfo;
int to_poll = 1, num_event, ev_num, i;

// block SIGIO and SIGRTMIN
sigemptyset(&sset); sigaddset(&sset, SIGIO); sigaddset(&sset, SIGRTMIN);
sigprocmask(SIG_BLOCK, &sset, NULL);

while (TRUE) {
    if (to_poll) {
        num_event = poll(ev_array, ev_num, 0);
        to_poll = 0;

        // call read or write handler if there is any event.
    } else {
        while (sig = sigtimedwait(&sset, &sinfo ,&wtime) > 0) {
            if (sig == SIGRTMIN) {
                if (sinfo.si_band & POLLIN)
                    read_handler(sinfo.si_fd);
                else if (sinfo.si_band & POLLOUT)
                    write_handler(sinfo.si_fd);
            } else if (sig == SIGIO) {
                // signal queue overflow, call poll() at next loop
                to_poll = 1;
                // clean signal queue
                while (sigtimedwait(&sset, &sinfo, &ts) > 0);
            }
        }
    }
}
```

Figure 4: Handling network events with RT signals

2.3 /dev/poll

/dev/poll[7] is first introduced in Solaris 7 in order to remove the need to specify interest set on every poll(). It is a state-based event dispatching mechanism. The idea behind it is that applications can open the device file to build a set of interesting descriptors inside kernel. This set is built gradually at every accept of a new connection, separating building of the interesting set from event retrieval. This reduces data copies of descriptor information between user space and kernel space. Interest sets inside kernel are built via writing pollfd structure into an open file descriptor on /dev/poll. ioctl() is used to poll all descriptors registered inside kernel, device driver's poll callback associated with each descriptor is called to get current state. Figure 5 explains how to use /dev/poll on a network server. /dev/poll is the first approximate implementation of Banga's declare_interest[2] idea.

Provos[10] implements this idea on Linux. The implementation caches latest results from device driver poll callbacks. Given a file descriptor, if there is no event between two ioctl() wait event call, previous cache result will be used for the second ioctl() and the device driver poll callback will not be executed. If there is any event, device driver marks corresponding cache entries dirty. At the next execution of ioctl(), dirty descriptors will be polled.

Because Provos's implementation of /dev/poll is state-based[1], there is no event collapsing or event queue overflow problem. /dev/poll also allows multiple interest sets in Linux kernel.

2.4 Summary

RT signals is event-based and events reported may not be the latest state change of descriptors. /dev/poll, like select() and poll(), is state-based and events reported is always the state of descriptors at the poll time. Both RT signals and /dev/poll are new scalable event dispatching mechanisms supported by kernel. However, these features are not available on all platforms. RT

[1]Libenzi[6] has a event-based implementation called /dev/epoll.

```
/* struct dvpoll {
 *     struct pollfd *dp_fds;
 *     int dp_nfds;
 *     int dp_timeout; } */

int i, num_event, dpfd;
struct pollfd fds[MAX_FD];
struct dvpoll dp = {fds, 0, 0};

dpfd = open("/dev/poll", O_RDWR);

while (TRUE) {
    num_event = ioctl(dpfd, DP_POLL, &dp);
    if (num_event == 0) {
        do_sleep(msec);
        continue;
    }

    for (i = 0; i < num_event; i++) {
        if (fds[i].revents & POLLIN)
            read_handler(fds[i].fd);
        else if (fds[i].revents & POLLOUT)
            write_handler(fds[i].fd);
    }
}
```

Figure 5: `/dev/poll` on a network server

features / methods	Scalable to large set of file descriptors	Event collapsing	Dequeue multiple event per system call	Event queue overflow	When queue overflow, kernel fallback to traditional poll()	Return initial state when declare interest fd	Multiple interest sets maintained in kernel for per process
select()	No	NA	Yes	NA	NA	NA	NA
poll()	No	NA	Yes	NA	NA	NA	NA
/dev/poll[1]	Yes	NA	Yes	NA	NA	NA	Yes
RT signals	Yes	No	No	Yes	No	No	Yes
RT sig-per-fd[1]	Yes	Yes	No	No	NA	No	Yes
declare_interest[2]	Yes	Yes	Yes	Yes	Yes	Yes	No

Table 1: Comparison of various event dispatching mechanisms

signals is officially supported since Linux 2.4 and `/dev/poll` is available only with kernel patch. Table 1 is a feature summary of all mechanisms discussed above. In Section 6, we will evaluate performance of event dispatching mechanisms available in Linux 2.4.

3 Proposed Solution: Temporal-locality-aware `poll()`

Most traffic of a TCP connection has the property of temporal locality. The most obvious example is HTTP connection. Users browsing the web have to think before clicking on next URL. HTTP requests arrive web server in bursty way. In addition, traffic in a single HTTP transaction is also bursty because of network congestion or protocol processing.

Based on this observation, we find that a file descriptor associated with a connection is idle most of the time. Either there is no event for a long time or there are many events in a short time. If a web server is loaded with many connections, most file descriptors in a single `poll()` loop have no event.

This means most of the `poll()` invoked by a web server is costly, no matter how few events there are. This tragedy is caused by the interface design of `poll()`. The two linear scans of a long descriptor event array, one inside kernel and one outside kernel, are the main performance bottleneck we want to overcome.

[1]Use of these mechanisms needs kernel patch. They are not available in Linux 2.4.
[2]`declare_interest`[2] is listed only for reference. There is no Linux ported version.

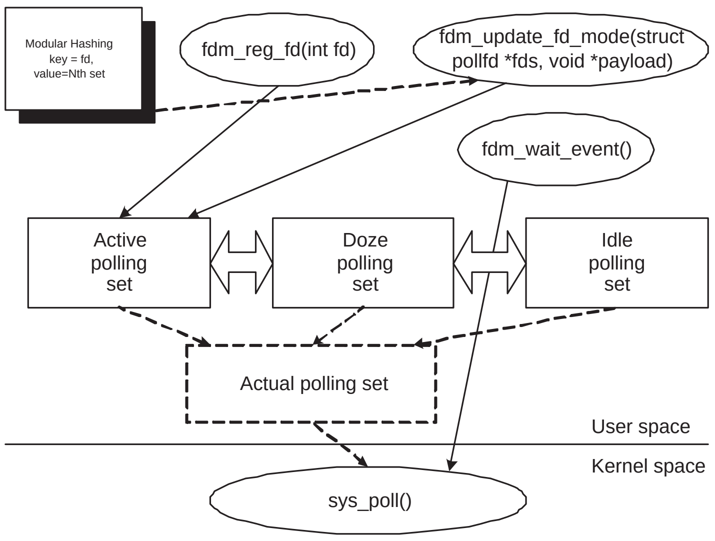


Figure 6: Architecture view of event dispatching library

The idea of temporal-locality-aware `poll()` is: Since a descriptor is idle most of the time, it doesn't need to be polled at every poll loop. It should be polled only when it is likely to have events recently. Events can be predicted based on the history of a descriptor. We approximate this by associate a counter with every descriptor. In a poll loop, the counter is increased if there are events, otherwise, it is decreased. All descriptors are divided among three polling sets according to this counter. The frequency of calling `poll()` to the kernel of each set is different. In this way, polling time on most idle descriptors is saved and active descriptors are checked more efficiently.

4 Proposed Library Interface

Our library is a thin layer sitting between server application and kernel system call. We named it FDM (File Descriptor Management Library), because it's goal is to manage all file descriptors in a server application. During the life time of file descriptors, file descriptors are kept in one of three polling set in the library, and are polled by library according to its live counter. Figure 6 illustrates the architecture of the library. Figure 7 is the function prototypes of the library.

```
typedef struct {
   int fd;
   short events;
   void *payload;
} fdm_event_t;    // data structure return by wait event

int fdm_start();  // library init
int fdm_stop();   // library shutdown
int fdm_reg_fd(const struct pollfd *fds, int lock);
int fdm_unreg_fd(int fd);

// tell library a fd is read interest or write interest
int fdm_update_fd_mode(const struct pollfd *fds,
                       void *payload,
                       struct timeval *tv);
int fdm_wait_event(fdm_event_t *ev_array,
                   unsigned int array_size,
                   int timeout);
```

Figure 7: Interface of event dispatching Library

In the library, interest set specification and event notification are separated. Server may register a file descriptor with `fdm_reg_fd()` once the descriptor is created by `accept()`. `fdm_reg_fd()` initially puts the descriptor specified in first argument into most frequently polling set. All registered descriptors will migrate between three polling sets depending on their respective live counters. The second argument specifies whether or not to lock the descriptor specified in the first argument from migration between polling sets. If a file descriptor is locked, it stays in most frequently polling set regardless of the value of its live counter. Listening sockets are usually being locked to achieve better performance. This concept is similar to *multi-accept*[3]. Because all connections and events of connections are derived from `accept()` of listening sockets, limiting the polling frequency of listening sockets will limit the performance of server. Besides, there is little temporal locality between events in a listening socket. Polling frequency of each polling set is discussed in next section.

`fdm_wait_event()` is used to receive network events. It generates a new polling set from three polling sets, invokes `poll()` with the new set as argument. `array_size` specifies the size of `ev_array`. Return events are put into `ev_array`. `timeout` specifies the time to wait if there is no event. Actually, timeout parameter is passed to `poll()` directly.

Given a file descriptor, server application may interest in reading it or writing it, but not both. For example, before server reads HTTP request from a file descriptor, it waits for read ready event of the descriptor. After it gets read ready event and reads HTTP request, it waits write ready event of the descriptor before writing HTTP reply. Server application must specify current interest (read or write) before calling `fdm_wait_event()`. This is achieved through `fdm_update_fd_mode()`. Current interest of a file descriptor is specified in `events` field of `fds` argument.

In general case, server application may need to maintain an array or a hash table to keep track of the association between a file descriptor and the threading or call-back data structure of a connection (These data structure are needed in a SPED server). When some events are returned from kernel, file descriptors of returned events are searched throughout the array for their threading or call-back data structure. Server needs a threading (or call-back) data structure to resume the execution of a thread (call-back function) that deals with a specific connection.

To remove the need for the server application to maintain another array and to eliminate the linear search mentioned above, the data structure for a file descriptors maintained in the library has a payload field that saves the address of associated threading or call-back data structure. Every event returned also carries a payload field, `payload` is a void pointer that can point to anything like threading or call-back data structure.

In addition, Server application may want to close a timeout connection if it is idle for a long time. This feature support is needed in this library in order to remove another time-stamp array (also the corresponding linear search) in server code. Both of the threading (call-back) data structure and the timeout time-stamp of a file descriptor can be updated through the arguments of `fdm_update_fd_mode()`.

`fdm_unreg_fd()` removes a file descriptor from polling sets of the library. This function is called when the connection is closed.

5 Implementation Details

5.1 Polling Frequency

Polling frequency will influence response time of a connection and overall throughput of the web server. `poll()` is invoked on every `fdm_wait_event()` call, however, not every polling set is polled. It depends on the value of default live counter. Assume default live counter is N, then active polling set is polled on every `fdm_wait_event()` call, doze polling set is polled on every N calls and idle polling set is polled on every N^2 calls. A file descriptor is downgraded from active to doze or doze to idle if there is no event for $N-1$ or N^2-N calls, and is upgraded from idle to active or doze to active if there are events for two successive calls.

A descriptor is initially put into active polling set once registered. If it has no event for $N-1$ `fdm_wait_event()` calls, it will fall into the category of second polling set, because second polling set assumes file descriptors inside it have events on every N `fdm_wait_event()`. The same logic holds true for descriptors downgrade from second polling set to third one.

Since `poll()` is a state-based event view, the time when a event occurs is not known. There is no way to determine event frequency of a file descriptor exactly. So the best way is to poll them on different frequency, let the descriptor falls into best category by itself. Although a more frequently `poll()` can be used to better predict frequency of events, this approach cannot be used since it conflicts with our goal to eliminate polling of idle file descriptors.

When a descriptor is not idle for two successive `poll()` calls, we upgrade it to active polling set immediately. This is because temporal locality property of network events. Other events for the same descriptor may arrive very soon. Our upgrade strategy keeps a fairly well response time for server application written on this library.

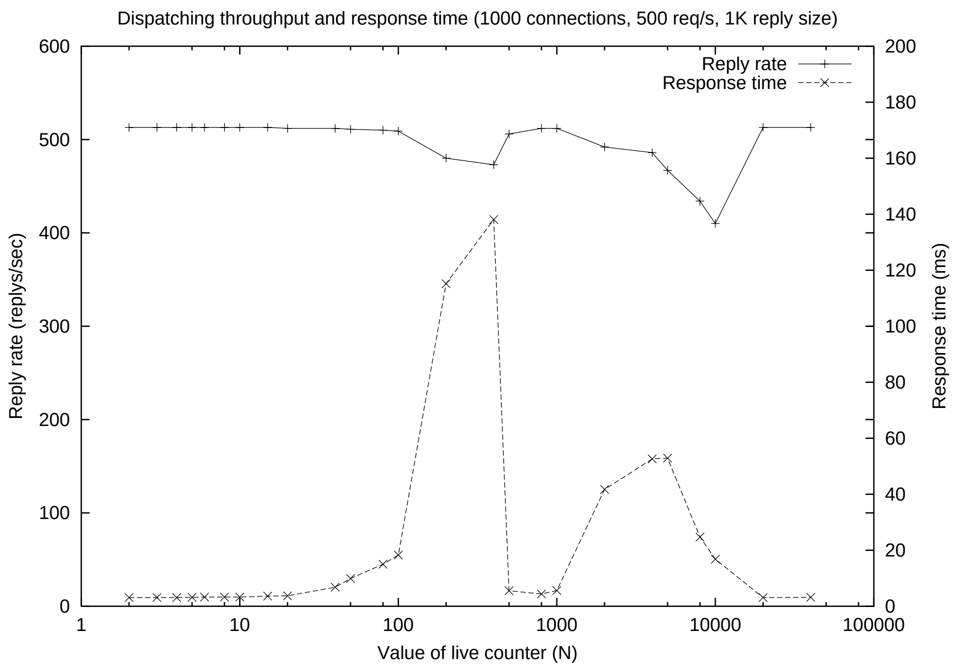


Figure 8: Server performance with different value of live counter

In our experience, when web server is overloaded, we found out that a small value of N maintains a good response time yet good improvement on throughput. A medium value of N further improves throughput but the response time is longer. A large value of N renders locality-aware `poll()` useless. Because most file descriptors will not downgrade and stay at active polling set, this results in long, large standard deviation of response time and worst throughput (some downgraded connections will suffer unacceptable response time). Our experience shows that a feasible N is between 3 and 10.

Figure 8 shows the influence of live counter on reply rate and response time of a memory-based web server. In this experiment, we establish 1000 persistent connections to the server. Each connection sends a request every 2 second, so that total request rate is fixed at 500 requests per second. This simulates a 1000 users browsing the web with 2 seconds think time. N influences request response time significantly. We take advantage of both, saving CPU time on polling more active descriptors and good response time, only at small N. With large N, temporal-locality-aware `poll()` approximates to plain `poll()`, file descriptors are not differentiated. Notice that response time and throughput doesn't always increase or decrease with the increase of N. The reason is: When $N < 400$, most file descriptors are fall into idle polling set; Increasing N may also decrease the polling frequency of idle set. This limits number of events a server can have, so reply rate drops too. However, when N is increased from 400 to 800, more and more descriptors in idle polling set are moving to doze polling set along with the increase of N, descriptor number in doze polling set is increased. Polling frequency is increased from every 400^2 wait event calls to 800 ones. So reply rate increases again. At $N > 800$, most descriptors falls into doze polling set, further increase of N decreases polling frequency, limiting number of events and make performance worse.

5.2 Multiple Threads Implementation

Gooch [5] mentioned the idea of dividing file descriptors among two threads. One thread polls mostly active file descriptors, another one polls the rest. However, multiple threads implementation is impractical. There are two reason. First, the overhead of synchronization and context-switching between threads will limit the performance of a server. Second, `poll()` is invoked automatically by all threads asynchronously. Polling frequency of each threads are determined by their respective sleep time, which is regulated by 10ms Linux timer. However, 10ms is so long that server is sleeping most of the time. This results in a long response time and low throughput on the server.

In fact, our library under the implementation of multiple threads has worse performance than a simply `poll()`. With a multiple threads implementation, polling frequencies of three polling sets are still under exponential relationship, limited by the same reason that `poll()` is state-based and event arrival time is unknown. It is not beneficial to implement this idea with multiple threads.

5.3 Web Server Sleep Threshold

Web server sleep threshold is the number of successive wait event calls without any event before web server sleeps. If the number of a sequence of successive wait event calls without any event reaches the threshold, web server is put into sleep.

When there are too few events to keep the server busy, put the server into sleep is often desired. However, as just mentioned, granularity of Linux timer is too coarse. If server is always put into sleep when there is no event. Server may sleep too much and have a lower throughput.

Sleep threshold is machine dependent. Because fast machine consumes events more quickly that it is easier to fall into the condition for sleeping. Fast machine needs to have larger sleep threshold to prevent limitation on the processing power. Sleep threshold is also event dispatching mechanism dependent. A scalable, fast event delivery mechanism like `/dev/poll` or RT signals needs to sleep less frequently too, for the same reason.

Notice that the value of live counter must be considered together with web server sleep threshold. Larger value of live counter generally decrease the number of descriptors to be polled, making event delivery faster too. In conclusion, a high performance web server requires fine tunes of sleep threshold by system administrator.

6 Performance Evaluation

This section shows the performance of memory-based web server on event dispatching mechanisms available in standard Linux 2.4, including our user-mode solution. We describe evaluation environment and implementation of our test web server. Server performance is evaluated on two metrics: dispatching overhead and dispatching throughput.

6.1 Evaluation Environment

To test various event dispatching mechanisms on a event-driven web servers, we modify *dphttpd*[6] to take advantage of our event-driven threading architecture. *dphttpd* is a very simple memory-based web server which does very simple HTTP protocol processing. It allows us to focus on performance of various event dispatching mechanisms without other constraints like disk I/O. Event-threading architecture provides both the advantage of easy programming and event-driven architecture to network servers implemented on it. It minimizes synchronization and kernel context-switching overhead by associating every connection with a user mode thread. Context switchings between user mode threads happen only when a user mode thread explicitly gives up control by waiting for I/O completeness or calling schedule yield function.

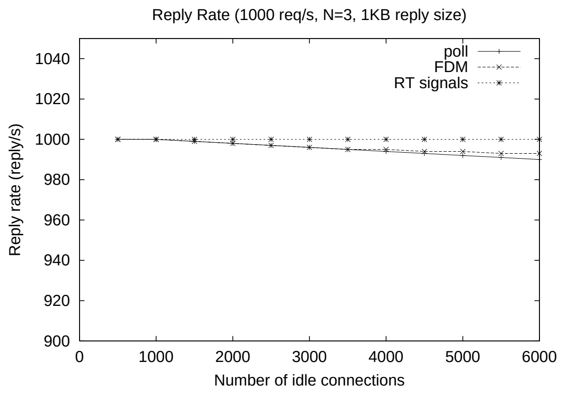
On our modified memory-based event-driven web server, three event dispatching mechanisms: `poll()`, temporal locality-aware poll and RT signals, all of them are available in standard Linux 2.4 without patch, are implemented. Parameters of the web server are optimized for each event dispatching mechanisms to be tested. Server sleep frequency is fine tuned for maximum performance. Our server does multiple `accept()` on a listening socket descriptor when `poll()` reports a ready event on the descriptor. This strategy minimizes the effect of the overhead on `poll()`. Because more connections are accepted and more useful works will be identified by follow-up `poll()`. Overheads of `poll()` are amortized(*multi-accept*[3]).

In the experiment, server runs on a Pentium III 600MHZ, 128MB RAM machine. Client loads are generated from three Pentium III 1GHZ, 512MB RAM machine with httperf[9]. Idle connections are made from a low-end machine. All machines are equipped with a Intel EtherExpress Pro 100 ethernet card, and connected together with a single 100Mb ethernet LAN.

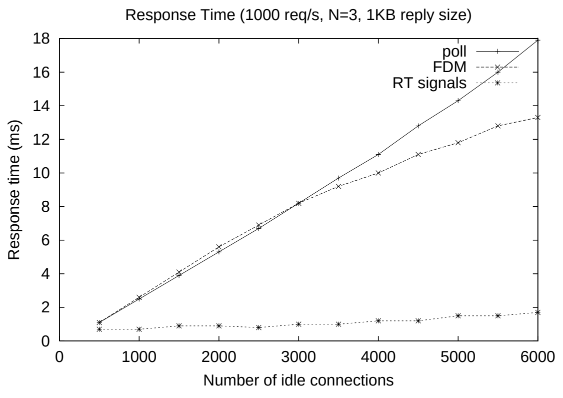
6.2 Dispatching Overhead

Dispatching overhead experiment simulates the condition when a web server is loaded with large number of slow, long distance, idle connections. To see the influence of these connections, web server is not overloaded. Request rates are fixed at comparatively light load. We measure the overhead of request handling as a function of number of idle connections.

Figure 9 shows the influence of such overhead on web server reply rate and response time. RT



(a) Reply rate



(b) Response time

Figure 9: Event dispatching overhead in terms of number of idle connections

signals mechanism provides a stable reply rate and very short response time. Although RT signals mechanism is scalable, idle connections still incurs small overhead on RT signals (Figure 9(b)). Because web server has to maintain an array of timeout stamp and `pollfd` data structure on every connection. On a regular web server, idle connections are timeout and disconnected when there is no activity for a period of time. To show the effect of overhead maintaining these arrays, code maintaining `pollfd` array and scanning through time-stamp array for timeout connections remains in our test web server, but timeout connections are not disconnected.

When the number of idle connections is less than 3000, FDM (temporal-locality-aware poll) has equal reply rate comparing to plain `poll()`, but a slightly longer response time. The reason is that the polling time (money) saved by FDM is less than the time (cost) spent on polling set management. Notice that request processing time on a plain `poll()` web server grows linearly with number of idle connections. When the number of idle connections is greater than 3000, the time wasted on idle connections for a plain `poll()` system is greater too. A large portion of time wasted in a plain `poll()` system can be saved in a event-locality-aware system. The time saved is greater than the time spent on polling set management, and is invested in active file descriptors by more `fdm_wait_event()` calls. This results in shorter response time for FDM comparing to the response

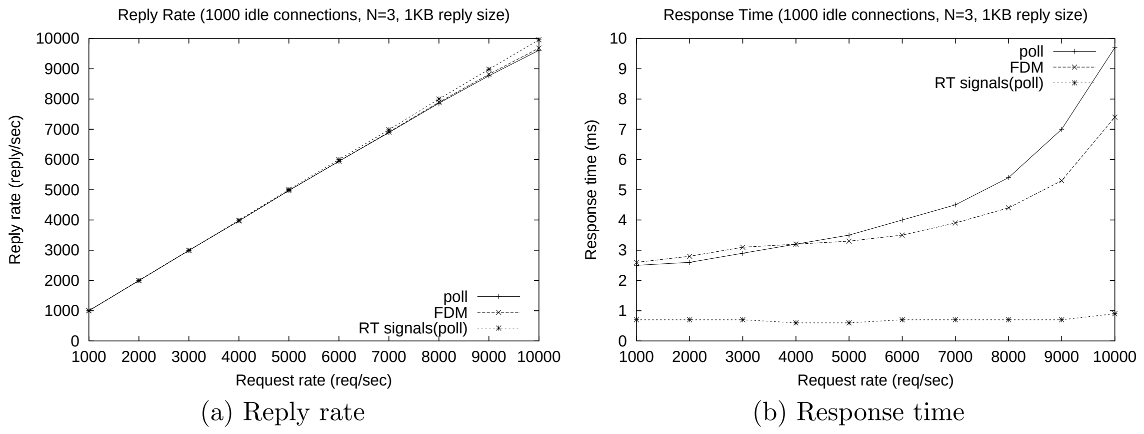


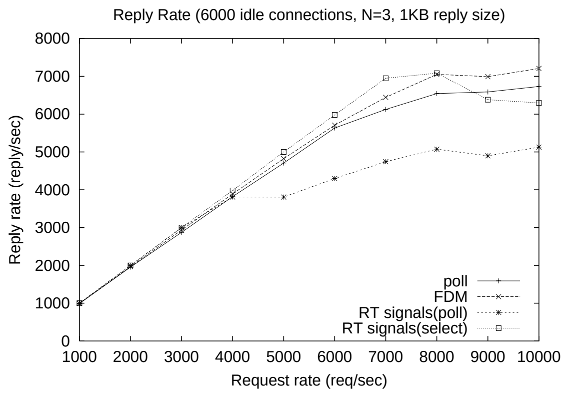
Figure 10: Server performance with 1000 idle connections

time for plain poll().

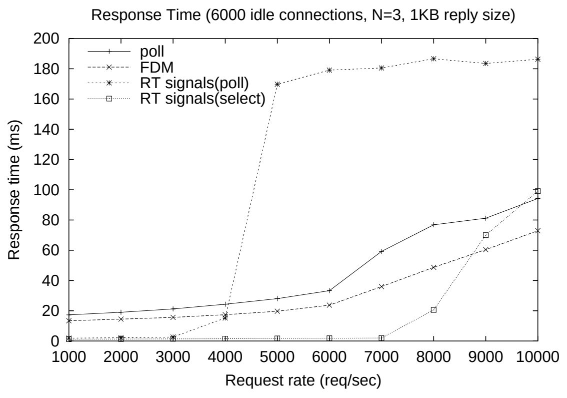
6.3 Dispatching Throughput

In dispatching throughput experiment, we want to see how much throughput a web server can achieve when the server is given a fixed number of idle connections. The performance is measured as a function of request rates. RT signals with poll() and RT signals with select() are evaluated respectively to see the problem of poll() when it is used to recover the server from RT signal queue overflow.

Figure 10 shows server performance with 1000 idle connections. Notice that all mechanisms scale pretty well with respect to the reply rate (Figure 10(a)). Request rate beyond 10000 requests per second is not feasible since throughput is limited by 100Mbit ethernet. Because we implement *multi-accept* on plain poll() test server, the reply rate of plain poll() is comparatively good. However, poll() is inherently not scalable to a large number of connections. Time spent on event detection is too long to maintain good response time when request rate increase. In Figure 10(b), you can see the response time of plain poll() increases with request rate. We also observe that the response time of poll() is a lot longer than that of RT signals even at light load, since overhead of 1000



(a) Reply rate



(b) Response time

Figure 11: Server performance with 6000 idle connections

idle connections exists for `poll()` no matter how request rate is. Under unscalable `poll()`, FDM improves response time with different polling strategy, but it can't change the fact that underlying mechanism costs time in proportion to number of connections. So response time of FDM increases with the increase of request rate.

To further observe the behavior of FDM library and RT signals, In Figure 11, we increase the number of idle connections to 6000. The advantage of FDM is clear and obvious. FDM-based web server delivers highest throughput and shortest response time.

However, it is surprising that the performance of RT signals(poll) falls behind even that of traditional `poll()` when request rate are greater than 4000 req/sec. The reason is because we use `poll()` to handle lost event when the signal queue is overflowed. To use `poll()` with RT signals, web server has to prepare and maintain a `pollfd` array for such signal queue overflow emergency. Rearranging this `pollfd` array on every several `sigtimedwait()` calls and the timeout time-stamp array is costly and costs CPU time in proportion to number of idle connections. At 6000 idle connections, RT signals web server spends much more time on the work we just mentioned than the time spent at 1000 idle connections. When client request rate is greater than 4000 req/sec, signal arrival rate become faster than signal dequeue rate. This results in many `SIGIO` signal queue overflow events and significantly increase of response time. This performance drop scenario can't be observed when server is loaded with just 1000 idle connection.

Though, RT signals web server can be improved by replacing `poll()` with `select()` to eliminate the need to maintain a `pollfd` array. Read, write, exception bitmaps parameter for `select()` allows direct access to the bit of every file descriptor. Thus, linear scan and maintenance of `pollfd` array are eliminated. The overhead of finding timeout connections can also be alleviated by reducing times of scan on timeout time-stamp array.

Figure 11 also shows the performance improvement of RT signals(select) over RT signals(poll). The reply rate and the response time of RT signals(select) is the best over all event dispatching mechanisms when request rate is smaller than 8000 req/sec. When request rate is greater than 8000 req/sec, excessively switching back and forth between RT signals and `select()` when RT signal queue is overflowed makes the performance drop. The performance of FDM also starts overtaking RT signals(select) at 8000 req/sec. RT signal queue overflow problem greatly influences the performance of both RT signals(poll) and RT signals(select) at different load.

7 Conclusion

In this paper, we've introduced various event dispatching mechanisms and describe the advantage and disadvantage of them. `select()` and `poll()` are suffered from scalability problem and long response time when number of connections is large. `/dev/poll` and RT signals are two scalable mechanisms available in Linux 2.4. `/dev/poll` caches latest poll result and do not poll a file descriptor again if its state doesn't change. RT signals mechanism scales well but has signal queue overflow problem. Our proposed user-mode library solution extends `poll()` by exploiting temporal locality property among events in a file descriptor. Each descriptor is associated with a live counter and polling frequency of a descriptor is determined by the counter. Since most web connections are idle in a regular web server, this reduces total number descriptors to poll and save more CPU time on useful work. We've shown that this scheme performs pretty well on a large number of idle connections. Another advantage of the library is the portability of server code.

Polling frequency of our library and web server sleep threshold must be considered together, since a fast event dispatching mechanism (wait event call finish more quickly) needs to have small sleep threshold to avoid limitation on performance. Change of live counter will influence the running time of each `fdm_wait_event()` call. In addition, both of the parameters influence server throughput and request response time. We do not yet find a optimal solution to determine these parameter online. It is our future work.

It is surprising that the performance of RT signals web server is not scalable when it is loaded with 6000 idle connections. This is because the excessively switching between the two mechanisms, `poll()`(or `select()`) and RT signals, when RT signal queue is overflowed. RT signals(poll) server has worse performance than RT signals(select) one. The overhead of maintaining `pollfd` array associated with `poll()` is very large and will slow down event processing speed. We conclude that `select()` should be used to protect RT signals when signal queue is overflowed.

Availability

The temporal-locality-aware poll library and related program mentioned in this paper is available at the following URL:

`http://arch1.cs.ccu.edu.tw/~lhr89/fdm/`

References

[1] Apache. The apache software foundation. `http://www.apache.org`.

[2] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, pages 253–265, June 1999.

[3] Abhishek Chandra and David Mosberger. Scalability of Linux event-dispatch mechanisms. In *USENIX Annual Technical Conference*, 2001.

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

[5] Richard Gooch. I/O event handling under Linux. `http://www.atnf.csiro.au/people/rgooch/linux/docs/io-events.html`.

[6] Davide Libenzi. Improving (network) I/O performance. `http://www.xmailserver.org/linux-patches/nio-improve.html`.

[7] Solaris 8 man pages for poll(7d). `http://docs.sun.com/?q=%2fdev%2fpoll&p=/doc/816-3330/6m9kamh9d&a=view`.

[8] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[9] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

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

[11] Signal-per-fd patch for Linux. `http://www.luban.org/GPL/gpl.html`.

[12] Squid. Squid web proxy cache. `http://squid.nlanr.net/Squid/`.