Understanding Event Dispatch Issues in Internet Server Applications

Tim Brecht
Hewlett-Packard Labs
Palo Alto, CA
brecht@hpl.hp.com

Michal Ostrowski
IBM TJ Watson Research Center
Yorktown Heights, NY
mostrows@us.ibm.com

Abstract

Several previous studies have examined techniques for improving Internet server performance by investigating and improving operating system support for event-dispatching mechanisms. These studies have been largely motivated by the commonly held belief that the overhead incurred in obtaining network I/O events was the main factor limiting Internet server performance. In this paper we evaluate the impact that the application's design and implementation can have on the server's ability to dispatch network I/O events.

Our experiments show that relatively minor and seemingly sensible modifications to the server's implementation can dramatically decrease both the peak throughput and the throughput obtained under overload conditions. We find that while it is important for a server to be capable of accepting connections at a high rate, it is perhaps just as important to ensure that the server is able to balance accepting new connections with making forward progress on existing connections. Consequently, we believe that comparisons between Internet servers aimed at evaluating I/O multiplexing mechanisms must consider the servers' workload-management architecture.

Our observations are used to implement a simple `select`-based micro web-server that dispatches events at surprisingly high rates. Under workloads designed to stress the performance of event dispatch mechanisms, our user-level `select`-based server's throughput exceeds that of the fastest event dispatching server reported in existing research literature [8] and is equal to or better than the Linux in-kernel TUX server [23]. Finally, we demonstrate that the insights from this paper can applied to other servers and event dispatch mechanisms by using them to improve the performance of TUX.

1 Introduction

Internet-based (or network-centric) applications have experienced incredible growth in recent years and all indications are that such applications will continue to grow in number and in importance. How operating systems support such applications is the subject of much activity in the research community, where it is commonly believed that existing implementations and interfaces are ill-suited to network-centric applications [5],[26], [21].

In many systems, once client demand exceeds the server's capacity and continues to increase the throughput of the server approaches zero. This is reflected in long and unpredictable wait times, or a complete lack of response for many clients. In other words, existing servers are not well-conditioned to load. It is precisely during these periods of high demand when being able to service customers may be most important to those who are relying on the server. Examples of such periods occur during sharp changes in the stock market, breaking news events, and the Christmas shopping season. Unfortunately, it is not practical or cost effective to provision all servers to handle peak demands because peaks demand can be several to hundreds of times higher than the average [2] [25].

Because modern Internet servers multiplex between large numbers of simultaneous connections, much research has investigated modifying operating system mechanisms and interfaces to efficiently obtain and process network I/O events [4] [5] [20] [21] [8]. In this paper we use a simple `select`-based Internet server to examine the impact that the application's design and implementation can have on the server's ability to dispatch events under conditions of heavy load.

We intentionally use `select` because its inefficiencies have been well documented [4] [20]. While we are not trying to argue that inefficiencies in `select` could not and should not be improved upon, we believe that too much emphasis has been placed on improving operating system implementations and interfaces rather than examining how applications can better interact with existing systems. Our goal is not to implement another high-performance web server but instead to concentrate on the interaction between the server and the operating system and on the application's ability to efficiently obtain and process network I/O events. Our experiments show that:

- Seemingly minor and sensible changes to the server

implementation can significantly impact its throughput.
- In addition to ensuring that new connections can be accepted at as high a rate as possible, it is equally important to ensure that the server spends time servicing existing connections.
- Contrary to popular belief, it is possible to implement a `select`-based server that is capable of dispatching events quite rapidly.
- Under workloads that stress event dispatching performance, our portable, user-level, `select`-based server obtains throughputs equal to or better than the TUX in-kernel Linux server.
- The insights obtained through our experiments can be applied to other servers; this is demonstrated by improving the performance of the TUX server.

This paper demonstrates that some of the cause of poor performance of `select`-based servers (and other event driven servers) is due not so much to inefficiencies in the implementation of and interfaces to `select` (and other event notification mechanisms) as to what could be viewed as an improper and unbalanced implementation of the server. As a result, comparisons between I/O multiplexing mechanisms need to ensure that the server or servers in which the mechanisms are being evaluated use the same approach to managing the server workload.

2 Background and Related Work

Current approaches to implementing high-performance Internet servers require special techniques for dealing with high levels of concurrency. This point is illustrated by first considering the logical steps taken by a web server to handle a single client request, as shown in Figure 1. Note that almost all Internet servers and services follow similar steps. To simplify our illustration, the example below does not handle persistent or pipelined connections (although all servers used in our experiments handle persistent connections).

1. Wait for and accept an incoming network connection.
2. Read the incoming request from the network.
3. Parse the request.
4. For static requests, check the cache and possibly open and read the file.
5. For dynamic requests, compute the result.
6. Send the reply to the requesting client.
7. Close the network connection.

Figure 1: *Logical steps required to process a client request.*

Several of these steps can block because they require interaction with a remote host, the network, a database or some other subsystem, and potentially a disk. Consequently, in order to provide high levels of performance the server must be able to simultaneously service partially completed connections and to quickly and easily multiplex those connections that are ready to be serviced (i.e., those for which the application would not have to block and wait). This may result in the need to be able to handle several thousands or tens of thousands of simultaneous connections [5] and to dispatch network I/O events at high rates.

The server implementation must also consider the strain that it will be placing on the underlying operating system and the efficiency with which it and the operating system are able to work in concert. Initial web server implementations handled concurrency issues by creating separate threads of control for each new connection and relying on the operating system to automatically block and unblock threads appropriately. Unfortunately, threads consume significant amounts of resources and server architects found that it was necessary to restrict the number of executing threads [12] [5].

More recent approaches to high-performance server design treat each connection as a finite state machine (FSM) with events triggering transitions between states. Connections are managed by multiplexing between different connections (FSMs). The server works on the connection on which forward progress can be made without invoking an operating system call that would block. This is done by tracking the file and socket descriptors of interest and periodically querying the operating system for information about the state of these descriptors (using a system call like `select` or `poll`). These calls indicate which operations can be performed on each descriptor without causing the application to block.

Significant research has been conducted into improving web server performance by improving both operating system mechanisms and interfaces for obtaining information about the state of socket and file descriptors [4] [19] [3] [5] [20] [21] [8]. These studies have been motivated by the belief that under high loads the overhead incurred by `select` (or similar calls) is prohibitive to implementing high-performance Internet servers. As a result they have mainly developed improvements to `select`, `poll` and `sigwaitinfo` by reducing the amount of data that needs to be copied between user space and kernel space or by reducing the amount of work required by the kernel (e.g., by only delivering one signal per descriptor in the case of `sigwaitinfo`).

Recent work by Chandra and Mosberger [8] introduced operating system modifications designed to improve web server performance. However, they found that a simple modification to a `select`-based web-server (with a stock operating system) outperformed the approach using their improved operating system and the improvements studied in prior research [20]. They refer to this server as a "multi-accept" server because upon learning of a request for a new incoming connection, rather than accepting a single connection, it attempts to accept as many incoming connec-

tions as possible. Calls to `accept` are repeated until there are no more outstanding connection requests or the limit on the maximum number of open connections is reached (limited by the number of open file descriptors permitted). The results of Chandra and Mosberger's experiments were contrary to conventional wisdom which believed that `select`-based servers perform poorly under high loads.

Our work in this paper is partially motivated by the work of Chandra and Mosberger [8]. Their work shows that even simple server designs exhibit a wide range of variation in performance that is not well understood. We believe that not enough emphasis has been placed on understanding basic Internet server design. Therefore, in this paper we consider a number of design options, study how the implementations interact with the operating system, and use these results to gain insight into some of the issues affecting server performance.

This paper differs from previous work in that we concentrate exclusively on the software architecture of the server. Specifically, we are interested in determining which aspects of a server's design contribute to, or help to prevent, server meltdown during periods of high load. The focus of much of this paper is on the interplay between how the server accepts new connections, obtains event information from the operating system, and uses that information to process existing connections. This approach provides us with new insights into techniques that can be deployed within the application to significantly improve performance. We believe that it is extremely important to understand these insights when designing, modifying or evaluating operating systems event mechanisms and interfaces.

3 Methodology

We examine a widely used and highly studied example of an Internet server, namely a web-server. We use the core of the Chandra and Mosberger [8] "multi-accept" server to create a new, highly parameterized micro-server. This permits us to easily explore a wide variety of implementation options and to study how the resulting implementations interact with the operating system and impact event dispatch performance. We use the multi-accept server as a core for our server because it provides the highest performance of all of the servers and event dispatching mechanisms considered in the Chandra and Mosberger study. As such, it is the fastest event dispatching server reported in existing research literature.

Our approach is both necessary and important. It ensures that any differences in performance are actually due to differences in the software architecture of the server and not due to other artifacts of the implementations being compared (e.g., differences in the caching algorithms, or numbers of file descriptors being used).

Unfortunately, it is not feasible to compare all combinations of parameters because of the exponential explosion in the number of experiments that would need to be run. Although we have not performed a completely systematic elimination of different combinations, we have explored the parameter space sufficiently to understand the behaviour of the server and to identify those combinations of options worthy of further consideration.

4 Server Implementation

Figure 2 shows the structure of our parameterized micro-server and provides a context in which we describe the various options and how they modify the server's behaviour. The pseudo-code contains some comments which describe how command line options affect the server implementation. It also contains annotations to show where the steps from Figure 1 are implemented (e.g., #2-5,7 represents steps 2 through 5 and step 7).

4.1 Server Parameters

We now provide a more complete list of options available for controlling how the server operates and explain how they are used to change the server's behaviour. Several of these options were included and implemented in order to faithfully recreate and then to further understand the performance of the Chandra and Mosberger [8] server. By default all options are disabled unless explicitly stated otherwise.

- `[--caching-on]` is used to turn caching on. Note that the real purpose of this option is to be able to eliminate the effects of file system accesses and to focus on the remaining system calls.
- `[--max-conns` *num*`]` sets the maximum number of simultaneous connections permitted. The default value used in all of our experiments is 15000 and all servers enforce this maximum. While this option is used to avoid running out of available file descriptors (i.e., no new connections can be accepted unless there are file descriptors available) this does not specify the number of permitted open file descriptors. Note that when caching is not used *num* may need to be less than one half of the maximum number of available file descriptors (since potentially each request could require two descriptors, one for the open socket and one for an open file). We have set the maximum number of open file descriptors to 32768 which easily accommodates our maximum of 15000 simultaneous connections.
- `[--accept-count` *num*`]` permits the server to consecutively accept up to *num* connections at a time. This is done by repeatedly calling `accept` until it either fails because there are no outstanding connection requests (setting `errno` to `EWOULDBLOCK`), until *num* new connections have been accepted, or until the maximum number of open connections has been reached. The default value for `num` is one. Zero indicates that there is no upper bound. The Chandra

```
server_event_loop() {
  while(1) {
    rdfds = readfds; wrfds = writefds;
    // find out which fds won't block
    n = select(...rdfds, wrfds...);

    while (fd = iterate_over_fds(n)) {
      if (ISSET(fd, rdfds)) {
        if (fd == accept_fd) {
          // connection requested
          // option [--accept-count]
          new_conns(accept_count);
        } else {
          // read, parse, process req.
          // when client done, read
          // returns EOF, close socket
          if ([--full-read] {
            // loop until failure
            while read_sock(fd);  #2-5,7
          } else {
            read_sock(fd);        #2-5,7
          }
        }
      }
      if (ISSET(fd, wrfds)) {
        // send result to client
        write_sock(fd);           #6
      }
    }
  }
} // server_event_loop

new_conns(int max)
{
  int fd, count = 0;
  while (count < max &&
         conns < max_conns) {
    // establish new connection
    fd = accept(listen_fd);       #1
    if (fd < 0) {
      break;
    } else {
      new_fd_init(fd);
      count++; conns++;
    }
  }
}
```

Figure 2: *The basic structure of the main server loop.*

and Mosberger [8] server uses no upper bound (i.e., `[--accept-count 0]`.

- `[--eager-read]` try to eagerly read from new connections. Call `read_sock` to read the request as soon as the new connection is accepted (i.e., from within `new_conns`). The server optimistically assumes that data will be available for reading when a connection is made on a new socket. If the assumption is incorrect, the `read` call simply returns with the error `EWOULDBLOCK`. Later `select` will indicate when the socket is readable.
- `[--eager-write]` try to eagerly perform writes to new connections when the response is available. Call `write_sock` to write the reply to the client's socket from within `read_sock`. Both `[--eager-read]` and `[--eager-write]` try to take advantage of any potential locality effects by working on the most recently used file descriptor and socket.
- `[--accept-on-close]` try to accept new connections after closing an existing connection. The motivation for this option is that if the server has reached the maximum number of simultaneous connections permitted and a connection is closed, then at least two file descriptors will be available and a new connection could be accepted. Recall that the number of permitted open files is more than twice the maximum number of simultaneous connections.
- `[--full-read]` specifies that the server should loop when calling `read_sock` until the call fails. This is included in order to completely recreate the behaviour of the Chandra and Mosberger [8] multi-accept server. They did this in order to consume all of the data in the socket and to take advantage of any locality affects. This approach is really designed for larger requests than those generated by `httperf` [15]. [1]
- `[--listenq` *num*`]` is used to set the server's listen queue length. The default is 128.

We provide this list of options to identify the range of parameters we have explored. For brevity and because they didn't significantly alter the results some options are not included here but are described in a technical report [7]. The caching option is only used when comparing performance with the TUX server in Section 7. This is because the Chandra and Mosberger server does not implement caching and because we found that although the system calls are exercised differently with the `[--caching-on]` option, the results obtained were not qualitatively different from those obtained without the option.

5 Environment

All experiments are conducted using a dedicated 100 Mbps Fast Ethernet switch that connects client hosts to our micro-server. The server executes on an HP NetServer LPr system running a Redhat 7.1 distribution with the Linux 2.4.2-2

[1] As we'll see the Chandra and Mosberger server does not do well under high loads. To be fair their server was originally designed to be used with RT-signals so the interaction with RT-signals and the need to avoid dead-lock is what motivated a number of their design decisions.

kernel. The processor is a 500 MHz Pentium-III with 16 KB L1 instruction and data caches and a 512 KB L2 cache. The system contains 320 MB of memory. The client load is generated using `httperf` [16] and ten B180 PA-RISC machines running HP-UX 11.0. The maximum number of open files was increased to 60000 for each of the clients.

For each data point in the graphs shown in this paper we start a new copy of the server. This is done because the server collects statistics of interest that we use in analyzing its behaviour under the specific load and because it permits us to collect `gprof` [10] statistics from each run. In experiments comparing the results obtained with and without `gprof` we found that using `gprof` did not qualitatively alter the results of our experiments. The largest difference we observed was a gap of about 4%. Although `gprof` is unable to apply accurate accounting techniques during interrupts (it simply adds the time spent handling the interrupt to the function currently being executed) we found the output of `gprof` quite instructive when viewed at a fairly coarse grain.

Each experiment is conducted by having the clients attempt to provide the desired load for a duration of 2 minutes using a time-out period of 3 seconds for each connection. The 2 minutes is sufficiently long to stress any of the system and application resources that must be limited, but short enough to permit us to conduct a reasonable number of experiments. Note that we have conducted several tests using significantly longer test durations to verify that 2 minutes is long enough to ensure that the servers and server queues have reached a steady state and that the test duration does not alter the results.

Any request that the server is unable to respond to is recorded by the client as an error. These errors may occur either because the server is unable to accept the connection or because it isn't able to provide a response before the client time-out period is exceeded. The client time-out of 3 seconds is also small enough to prevent retries if a connection can't be established.

Unless otherwise noted, all client requests are for the same one-byte file. This is done in order to place as much stress as possible on the web-server and the underlying operating system. This type of workload, one in which a single small file is repeatedly requested, is one which is commonly used in order to evaluate event dispatch mechanisms [20] [21] [8] [17]. While other workloads are of interest to us and will be considered in future work, they are not capable of placing the desired strain on the operating system and server. Note that we do include results for 2 KB files when comparing the performance of our server with TUX (in Section 7). In this case the file size is limited to 2 KB to prevent the 100 Mbps network from becoming a bottleneck. No warm-up period is used and the entire workload clearly fits in memory.

We have modified the default maximum number of open files permitted on the server to 32768. This is done using `/proc/sys/fs/file-max`. To accommodate the increased number of possible open files we have also increased the size of an `fd_set` by modifying the definition of __FD_SETSIZE to 32768.

6 Server Variations and Experiments

In this section we conduct a number of experiments to evaluate the performance obtained using our web-server with several different options. This approach permits us to explore a variety of web-server implementations, how they interact with the operating system, and to compare their performance. Table 1 lists the servers discussed in the upcoming sections, provides a quick reference for options used in each case, and describes how the combination of options modifies the behaviour of the server.

6.1 Basic Configuration Alternatives

We begin by contrasting the implementation of a relatively simple server with that of the multi-accept server used in the Chandra and Mosberger study [8]. We then compare the throughput obtained using these servers under conditions of heavy load with that of a new proposed server implementation.

The first server we examine might be considered a simple or classic `select`-based server since it is a design that is typical of what most people might implement. It makes no attempt to be clever about how to process requests, relying on `select` to obtain all information about what state each of the file descriptors of interest is in and then taking the appropriate actions. It is also a design that has been used in previous research [4] [5] and as a result `select` has been dismissed as being too inefficient to use in high-performance servers. We refer to this server as the Accept1 server because each time `select` indicates that the listening socket is readable (meaning that a new connection has been requested) `accept` is called once to accept a new connection. This server is implemented using all of the default options of our parameterized server, the most relevant of which is the `[--accept-count 1]` option. Recall that all servers use a default of `[--max-conns 15000]` to permit a maximum of 15000 simultaneous connections.

We compare the Accept1 server with the server used by Chandra and Mosberger [8] (referred to as OrigCM, for original Chandra and Mosberger). This server is used as a basis for comparison as it is the best performing server reported in previous research. We compare this server with our server configured to implement the same behaviour as the Chandra and Mosberger server (this version is called CM for Chandra and Mosberger). We include these two versions of the same server to demonstrate that our server faithfully implements the original Chandra and Mosberger server.

Name	Options	Server Behaviour
Accept1	`--accept-count 1`	A simple server that uses all of the default options. It is representative of what most people would likely implement and of the type of server that has been used in previous research [4] [5]. This server is named Accept1 because one of its most important characteristics is that it accepts only one new connection each time `select` indicates that the listening file descriptor to readable.
AcceptInf	`--accept-count 0`	This server repeatedly attempts to accept new connections when `select` indicates that the listening socket is ready (readable) with no limit on how many new connections it will accept. It only exits the accept loop if either there are no more incoming connection requests (the `accept` call fails and sets the `errno` to `EWOULDBLOCK`) or the maximum number of simultaneous connections have been opened.
Accept25	`--accept-count 25`	Accept a maximum of 25 new connections when `select` indicates that the listening socket is ready (readable).

Eager	`--accept-count 0`
	`--eager-read`
	`--eager-write`
		This is the AcceptInf server with eager reads and writes. This server eagerly tries to make as much forward progress as possible on each new connection by immediately trying to read from and write to the new connection.

Eager+full	`--accept-count 0`
	`--eager-read`
	`--eager-write`
	`--full-read`
		This is the Eager server with the addition of trying to fully read from a new connection. That is, immediately after the initial read from the new socket is performed subsequent reads are performed until the read call fails.

Eager+close	`--accept-count 0`
	`--eager-read`
	`--eager-write`
	`--accept-on-close`
		This is the Eager server with the addition of trying to accept one or more new connections whenever an existing connection has been closed.

CM (Eager+full+close)	`--accept-count 0`
	`--eager-read`
	`--eager-write`
	`--full-read`
	`--accept-on-close`
		This is the set of options required to implement the behaviour of the original server used by Chandra and Mosberger [8]. It is the Eager server with both options to fully read from new connections and to accept new connections when an existing connection is closed. Note that using our naming scheme this server could also be called Eager+full+close.

OrigCM		This is the original version of the Chandra and Mosberger server.

Table 1: *List of servers tested, options used to create them, and how the options affect their behaviour.*

The options required to implement CM reveal the key features of the Chandra and Mosberger server. The `[--accept-count 0]` option means that it aggressively accepts as many connections as possible when `select` indicates that the listening socket is readable. New connections are accepted until either there are no more outstanding connections to accept or the maximum number of open connections is reached. It attempts to make as much forward progress as possible on each new connection by eagerly trying to read from the socket immediately after a connection is accepted (`[--eager-read]`) and then immediately writing the reply to that socket (`[--eager-write]`). The `[--eager-read]` option optimizes for the case when there is data waiting on the socket by the time the server accepts the new connection and the `[--eager-write]` option continues working on the most recent connection in a hope to complete the request and capitalize on potential locality effects (by working on the most recently used file descriptor and socket). These options avoid checking with `select` to determine the state of the socket associated with the new connection. The `[--accept-on-close]` option is used to ensure that if the server enters a mode where the maximum number of open connections is reached, new connections will be accepted when the server finishes processing existing connections. The `[--full-read]` option is used to try to consume all of the data available on the socket and to take advantage of any locality effects. This is really designed for use with larger requests than those generated by `httperf` (e.g., if the socket buffer contains more data than is being read by a single call to `read`). The design of the Chandra and Mosberger server was heavily influenced by its original use of RT-signals and a number of the design choices were made to avoid dead-lock [15].

Although portions of this experiment look similar to one of the experiments conducted in the Chandra and Mosberger paper there are two important differences. First, our Accept1 server differs from their implementation of the `select` server because their version uses eager reads and writes where our version does not. Second, we consider loads that are significantly higher than the 5000 requests per second used in their paper.

Figure 3 shows the throughput obtained using the Accept1, OrigCM, CM, and AcceptInf servers as a function of the request rate. We used these server versions as a base for our comparisons and examine refinements later in the paper. Since we are particularly interested in the behaviour of these servers under overload conditions, we consider request rates of more than double the server's saturation point.

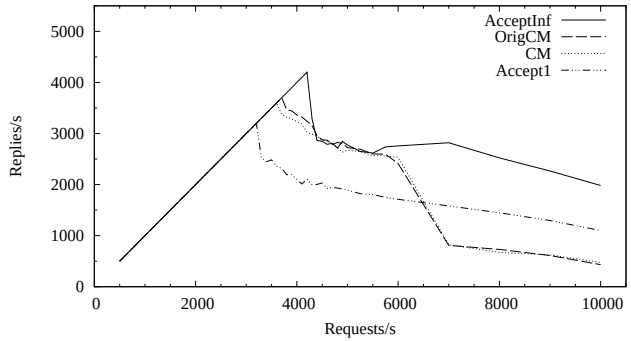

Figure 3: *Comparing the Accept1, CM, OrigCM and AcceptInf servers.*

We first point out that the throughput of the CM server quite closely matches that of the OrigCM server over the full range of loads used in our experiments. This indicates that the parameterized version of the server does faithfully implement the original Chandra and Mosberger server and that the overhead incurred by collecting statistics is negligible. For the remainder of the paper we use the CM server, since our server collects a number of statistics that can be used to obtain insights into the server's behaviour.

While the CM server provides better throughput than the Accept1 server for request rates between 3100 and 6000 requests per second, its throughput drops significantly and is quite poor for higher request rates. At a load of 6000 requests per second 5% of the calls to read data from a socket fail because there is no data. The `read` call returns with an `errno` of `EWOULDBLOCK`. With 7000 requests per second this jumps to 26% which we believe is the main cause of the drop in throughput. Under this load the `read` system call accounts for 21 seconds of the total 120 seconds of execution time. While some of these calls are to obtain cached file data, significant effort is wasted on ineffective system calls attempting to read from sockets that do not contain any data. We conduct an additional sequence of experiments in Section 6.2 to provide other insights into the performance of the CM server.

More interestingly, the AcceptInf server provides significantly better throughput than the other servers. It obtains a peak throughput of 4200 replies per second compared with 3600 for the CM server, an improvement of 17% (14% when compared with OrigCM). Additionally, under a load of 7000 requests per second or higher, the AcceptInf server's throughput is nearly 4 times that of the CM server and almost twice that of the Accept1 server.

Figures 4 and 5 show `gprof` output for the Accept1 and AcceptInf servers, respectively. These graphs show how the time spent in various routines changes over time. All of the routines that contribute to the major portion of the execution time are shown. Note that those routines that individually account for a very small percentage of the execution time are not included (typically 2 or 3 % each). Cumulatively they account for the missing percentage of the times (i.e., this is why the values in the graphs don't sum to 100).

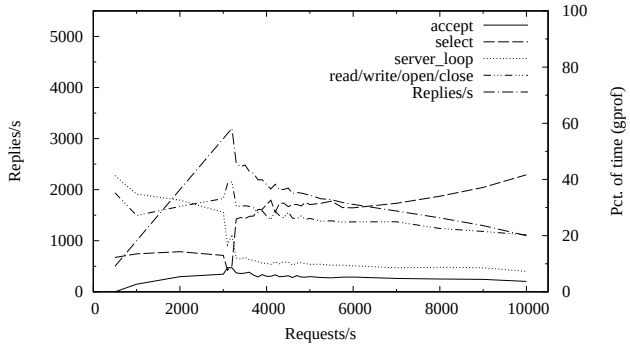

Figure 4: *Changes in where the Accept1 server spends its time as the load increases.*

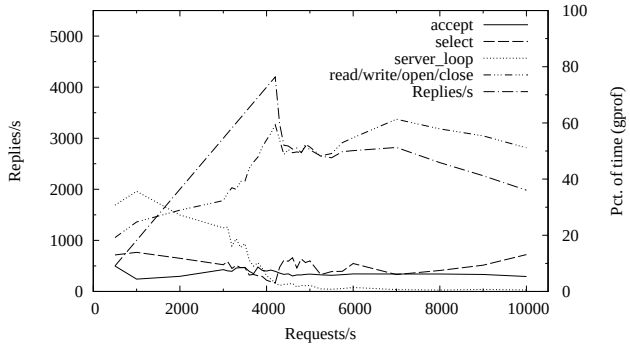

Figure 5: *Changes in where the AcceptInf server spends its time as the load increases.*

Most notable in Figure 4 is the percentage of execution time the Accept1 server spends in the `select` call after the server reaches its saturation point. It goes from 13% when the request rate is 3000, to 8–9% when the request rate is 3100 and 3200, to 26% at 3300 and over 40% at 10000 requests per second. Figure 4 shows that over this

range of request rates, the changes in the time spent making forward progress on existing connections (i.e., performing `read`, `write`, `open`, and `close` system calls) reflects almost the opposite of the changes in the time spent in `select`. This server is spending too much of its time in the `select` system call relative to the other work that it could and should be doing. A similar observation was also made by Banga and Mogul when using their unmodified kernel [4]. Rather than improving the kernel implementation as Banga and Mogul have done, we work with the existing operating system implementation by exploring techniques for improving performance while only modifying the application. Figure 5 shows how our AcceptInf server spends a significantly lower percentage of its time in `select` even under very high loads. As a result, it is able to spend more time making progress on existing connections (i.e., doing `read`, `write`, `open`, and `close` calls) and performs significantly better than the Accept1 server.

These results demonstrate that, as also pointed out by Chandra and Mosberger, considerable improvements in server throughput can be obtained by accepting multiple connections. This greatly reduces the time spent in `select` and permits the server to spend more time handling requests. However, we have demonstrate that the CM server's performance seriously degrades as the load increases and that our AcceptInf server provides significantly higher throughput.

6.2 The Impact of Eager Reads and Writes

In the next set of experiments our goal is to determine what aspects of the AcceptInf server help it to obtain its high peak throughput, and to help us better understand the characteristics that are required of a server capable of dispatching events at high rates. In this section we describe and compare the Eager, Eager+full, Eager+close, and CM servers.

The operation of the Eager server is identical to the AcceptInf server except that Eager attempts to eagerly read from and write to new connections (`[--eager-read]` and `[--eager-write]`). The Eager+full server behaves the same as the Eager server with the addition that all read attempts are repeated until they fail (`[--full-read]`). The Eager+close server differs from the Eager server only in that it attempts to accept new connections each time an existing connection is closed (`[--accept-on-close]`). Note that the CM server could also be thought of as an eager server that does full reads and attempts to accept new connections when completed connections are closed. Using our naming scheme it could also be called Eager+full+close.

Figure 6 shows that the throughput of the Eager+close server is significantly worse than the AcceptInf, Eager, and Eager+full servers. This indicates that it is the attempts to accept new connections whenever an existing connection is closed that limits the Eager+close server's throughput. The problem is that this server is extremely aggressive in attempting to accept new connections, doing so both when `select` indicates that there is an incoming connection request and whenever an existing connection is closed. Note that this is also the main reason the CM server's throughput is limited under high loads. Figure 7 shows how the `gprof` output for the Eager+close server changes as the load is increased. Note that under heavy loads the Eager+close server spends much of its time either accepting new connections or calling `read`, `write`, `open`, and `close`. This means that it is making decisions about what work to do with very little reference to information from `select`.

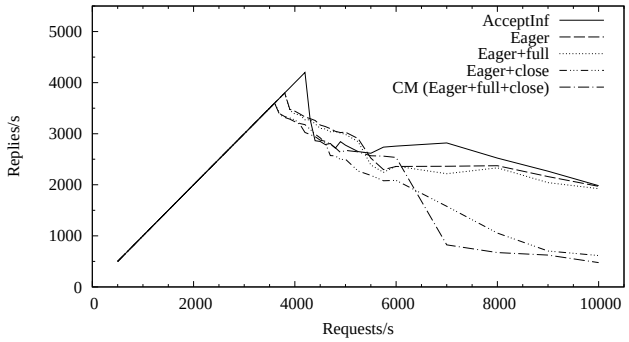

Figure 6: *Comparing the AcceptInf and Eager variations of servers.*

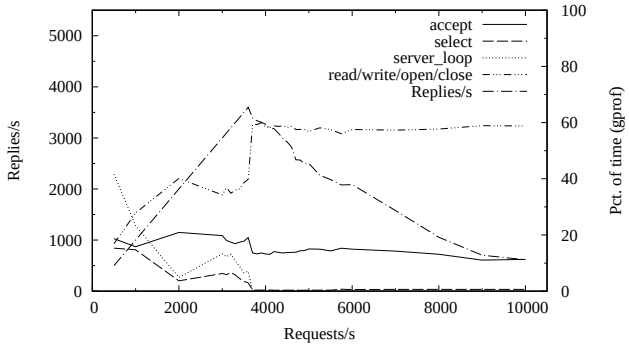

Figure 7: *Output from gprof for the Eager+close server.*

When comparing Figure 7 with `gprof` output for the AcceptInf server in Figure 5, we see that the Eager+close server spends 15–20% of its execution time accepting new connections while the AcceptInf server spends only about 5–10% of its time in the `accept` call. The AcceptInf server accepts considerably more connections using a significantly smaller portion of the server's execution time.

When the imposed load is 4000 requests per second, 93% of the calls to `accept` in the Eager+close server successfully accept a new connection. The average number of iterations of the loop within `new_conns` is less than one. This is because the check against the maximum number of open connections is contained in `new_conns`. Un-

der the same load, the AcceptInf server successfully establishes a new connection with a client in 88% of the calls to `accept`. In this case an average of 7 connections are accepted per call to `new_conns`. The AcceptInf server is doing a better job of ensuring that new connection requests are being satisfied thus permitting it to obtain higher peak throughput.

Figure 6 also shows that the Eager and Eager+full servers exhibit nearly identical performance. This indicates that for the Eager+full configuration of the server the `[--full-read]` option doesn't significantly impact performance. However, the `[--full-read]` option does appear to have a fairly significant impact on the server's performance if used in combination with the `[--accept-on-close]` option (as is the case for the CM server). When comparing the performance of the CM server (recall that this could also be called the Eager+full+close server) with that of the Eager+close server, we see that the CM server provides higher throughput in the 5000 – 6000 request per second range while the Eager+close server outperforms the CM server when the load is in the 7000 – 10000 range. These results are an excellent example of how seemingly sensible and minor modifications to a server can significantly impact its performance.

Figure 6 also shows that although the AcceptInf server has higher-peak throughput than the Eager server, the Eager server does obtain higher throughput for loads in the range of 4300 – 5250 requests per second. To better understand the reasons for this we show the `gprof` output for the Eager server in Figure 8. By comparing this graph with the graph showing the `gprof` output for the AcceptInf server (Figure 5), we see that the AcceptInf server's peak throughput is obtained by spending more time in `read`, `write`, `open`, and `close` and less time in `accept` than the Eager server. However, in the range of 4300 – 5250 requests per second the Eager server obtains better throughput. We believe this is because under these loads the Eager server is doing a better job than the AcceptInf server at making forward progress on existing connections (experiments in Section 6.3 will support this claim). In other words, the AcceptInf server is placing too much emphasis on accepting new connections and not enough on making forward progress on existing connections. For example, at a load of 4800 requests per second the Eager server spends about 55% of its time working on existing connections (i.e, in `read`, `write`, `open`, and `close`) compared with AcceptInf under the same load, which spends only about 50% of its time working on existing connections. In the next section we employ a technique that provides us with the best of both of these servers.

6.3 Limiting New Connections

Neither the AcceptInf server nor the Eager server discussed in the previous section is very satisfying. The Eager server suffers from a comparatively lower peak throughput in favour of a much smaller drop in performance once the saturation point is reached. On the other hand, the AcceptInf server suffers from a significant drop in throughput once its saturation point is reached but benefits from a much higher saturation point. In this section we examine a technique for trying to ameliorate the severity of this drop in throughput while still maintaining high peak throughput. In a sense, we attempt to devise a server that is a compromise between these two approaches.

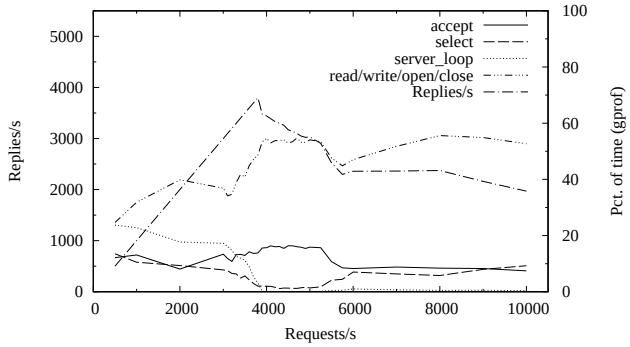

Figure 8: *Output from gprof for the Eager server.*

Our approach is to use the AcceptInf server in such a way that we limit the number of consecutive connections that are accepted thus placing less emphasis on accepting new connections and more on making forward progress on existing connections. For a first approximation of a limit we observe, from server statistics, that when the throughput peaks at 4200 requests per second the maximum number of consecutive connections accepted by the server was 104. We then conducted a series of experiments with limits on the number of consecutive accepts of 10, 25, 50, 75, 100, and 125. We explored limits that are mainly lower than the maximum to increase the likelihood of obtaining preferred average behaviour as opposed to trying to operate at the maximum level. We found that a server that limited the number of consecutive accepts to 25 (`[--accept-count 25]`) resulted in good peak throughput and did a better job of avoiding the drop in throughput that limits of 10, 50, 75, 100, and 125 experienced after the saturation point was exceeded. The peak throughput obtained using the server with a limit of 10 was lower than that obtained with servers using the other limits because it is not aggressive enough in accepting new connections. Therefore, we continue our experiments and discussion with a server that uses a limit of 25. We refer to this server as the Accept25 server since it repeatedly calls `accept` when `select` indicates that the listening socket is readable but stops when either 25 new connections have been consecutively accepted, the maximum number of open connections is reached, or there are no more outstanding connection requests.

The graph in Figure 9 compares the throughput of the Accept25 server, with the throughput obtained using the

AcceptInf, Eager and CM servers. The results of these experiments show that the Accept25 server is able to obtain peak performance equal to that of the AcceptInf server. More interestingly, the drawbacks resulting from the excessive emphasis that the AcceptInf server places on obtaining new connections are ameliorated by limiting the number of new connections. As a result, throughput improves significantly when the request rate is just slightly higher than the saturation point. Not only is throughput significantly better than with the AcceptInf server under loads of 4300 – 5500 requests per second but it is also significantly better than that obtained using the Eager server. This shows how important it is for the server to carefully balance the work it expends accepting new connections (which is necessary in order to obtain high peak throughput) and making forward progress on existing connections (which is also necessary to obtain high peak throughput, but appears to be more important for avoiding dramatic decreases in throughput under overload conditions).

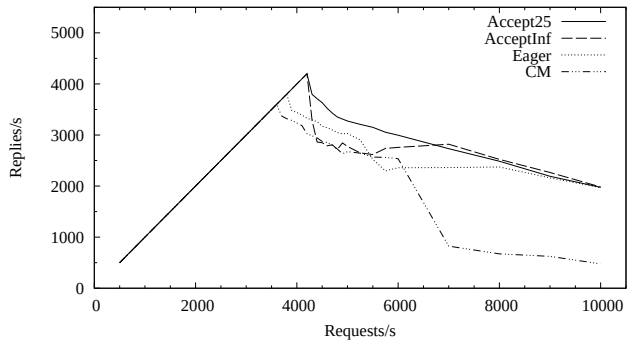

Figure 9: *Limiting the number of consecutive connections the server permits and the influence on server throughput.*

To more clearly understand how the different servers end up working with different numbers of conections Figure 10 shows how the average number of open connections (shown using a log scale) varies by server and changes with load. The average number of open connections is computed by sampling the number of open connections each time a new connection is accepted. Note that we have ensured that the servers have all executed for long enough that a steady state behaviour has been reached (i.e., significantly longer executions result in statistically similar numbers of open connections and throughputs).

The Accept1, Accept25, and AcceptInf servers can be compared directly since they behave in identical fashions, except for the number of connections they accept when `select` indicates that there is an outstanding connection request. As would be expected from examining their performance under heavy loads, the Accept1 server maintains the fewest number of open connections (too few for good throughput) and the AcceptInf server maintains the largest number of open connections (too many for good throughput). In contrast, the Accept25 server is able to operate

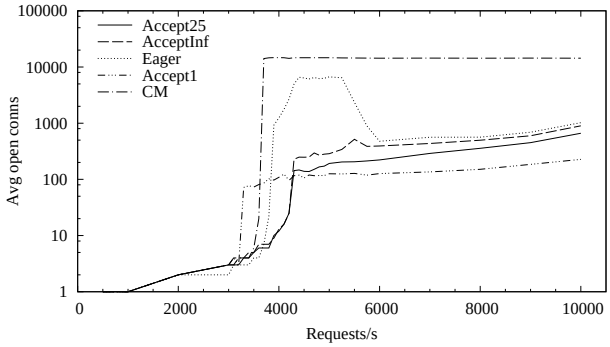

Figure 10: *The average number of open connections as the load increases.*

with a number of open connections that is part way between these two servers and as a result is able to achieve higher throughput.

This graph also shows more clearly that under high request rates the CM server attempts to work with far too many connections simultaneously (the result of an overaggressive approach to accepting new connection). As a result it reaches the maximum number of permitted connections (15000), in which case the server is only able to accept new connections when existing connections are closed. The Eager server also operates with a larger number of open connections than the Accept1, Accept25 and AcceptInf servers. This is most noticeable in roughly the 4000 to 6000 requests per second range. This is because the Eager server oscillates between a phase of working aggressively to make forward progress on existing connections and a phase of accepting large numbers of connections that have queued in the meantime. When the incoming request rate exceeds about 5250, the time between accept phases becomes too large to be able continue to accept connections at the rate being generated and the number of open connections is reduced.

An ideal server would sustain throughput equal to its peak throughput while the load continues to increases past the server's saturation point. One of the reasons that these servers (in particular the Accept25 server) are not able to sustain their peak response rates is because of the time the kernel spends handling TCP SYN packets that end up being discarded because the TCP SYN queue is full. A TCP SYN packet is the first packet received from a client that is initiating the three-way handshake required to establish a TCP connection.

Figure 11 shows both the response rate and the rate at which the kernel drops incoming TCP SYN packets (QDrops/s) when running the AcceptInf and Accept25 servers. [2] As mentioned previously, the size of the queue used in all of our experiments is 1024.

[2] Martin Arlitt [1] recently discovered that the version of the Linux kernel used in these experiments over counts the number TCP SYN packets dropped. This results in counts (rates in the graph) that are roughly double the actual values.

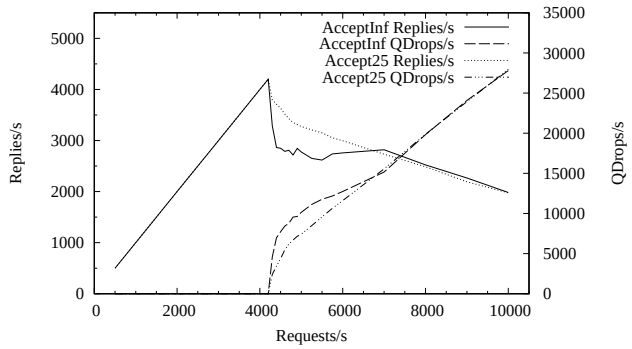

Figure 11: *Reply rates and TCP SYN queue drop rates.*

Figure 12 shows the mean response times observed using the servers discussed in this section as a function of the request rate. Note that the mean recorded response times are only accurate to one millisecond and are shown using a log scale. This graph is shown to demonstrate that the increases in throughput are obtained without significant sacrifices in mean response times.

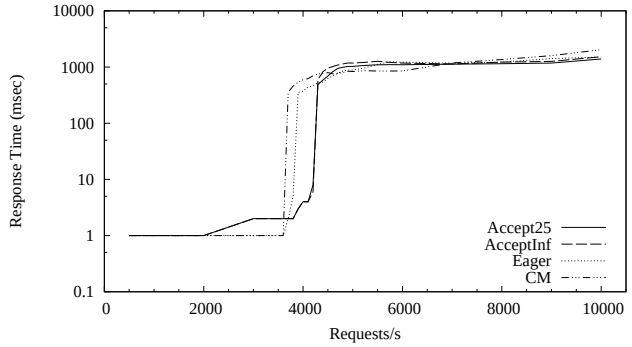

Figure 12: *Comparing the mean response times for different server options while increasing the load.*

7 Improving the Performance of TUX

Out of curiosity we decided to compare the performance of our best performing server configuration with the performance of the Linux kernel-level TUX server [23] [13]. We were interested in finding out how much better throughput TUX would obtain since events are dispatched within the kernel as they arrive and no system calls or context switches are required. Additionally, several optimizations are implemented in TUX, some of which eliminate significant amounts of data copying [23] [13].

We have tried to ensure that the two servers execute with comparable configurations in terms of resources. Both use a `listenq` of 128, support a maximum of 15000 connections, disable the Nagle algorithm, and use a single thread. Because caching is enabled in the TUX server we've also enabled caching in our Accept25 server. This server is referred to as the Accept25+$ server ($ represents cache). Figure 13 shows the throughput obtained using the Accept25+$ server, the TUX server, and a server labeled TUX25 (described shortly).

Interestingly, the TUX server did not dispatch events at the rate we had expected. The Accept25+$ server obtains throughputs as high or higher than the TUX server across the full range of server loads for both 1 byte and 2 KB requests (labeled 1B and labeled 2K respectively). Perhaps more interesting is the significant drop in throughput the TUX server experiences under the 1 byte workload, when the request rate exceeds the server's saturation point. This drop occurs because, as we'll demonstrate shortly, the TUX server is trying too hard to accept new incoming connections and is not spending enough time processing existing connections. We expect that the reason the TUX server's performance is able to improve as the request rate increases is because eventually the maximum number of connections is reached (15000). Once this limit is reached it is able to concentrate on satisfying existing requests.

Note that these graphs should not be viewed as demonstrating that the Accept25+$ server outperforms the TUX server. We believe that what they do demonstrate is how important it is for the server to balance (or schedule) the work that it is performing. In order to further demonstrate this point we show how our insights can be applied to improve the throughput of the TUX server for these workloads. We have made a minor modification to the TUX server to attempt to constrain the number of consecutive connections accepted to 25 (we call this version TUX25).

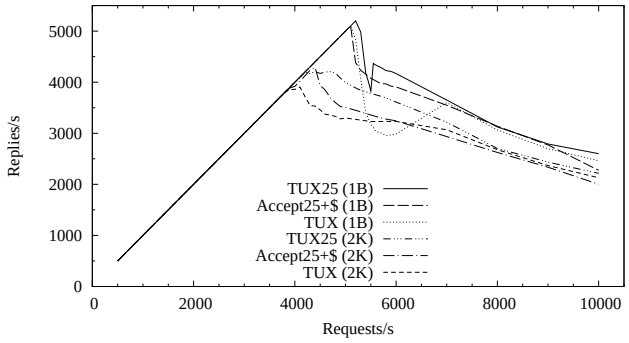

Figure 13: *Comparing the throughput obtained with our best server configuration Accept25+$, the TUX server, and our improved version of TUX (TUX25).*

Figure 13 shows that this version of the server offers significant improvements over the original TUX server for both 1 byte and 2 KB workloads.[3] The improved TUX server has 40% higher throughput at 6000 requests per second for 1 byte files and 25% higher throughput at 4700 requests per second for 2 KB files. While a limit of 25 may not be the optimal value (we haven't tried other values

[3]We don't know why the performance of the TUX25 server dips in the 1 byte request case at around 5500 requests per second but the results were repeatable.

yet) we believe that it clearly demonstrates the importance of ensuring that the server provides a reasonable balance between accepting new connections and making forward progress on existing connections. It also demonstrates that this balance is important even when using a different server and a radically different approach to dispatching events.

Our experiments in this section also highlight the importance of ensuring that the application servers being used to compare event dispatch mechanisms behave similarly with respect to their approach to workload management. Although the Accept25+$ server was able to obtain higher throughput than the TUX server, this was due to its ability to better balance its work *not* because it can more efficiently dispatch events. Comparisons between different event dispatch mechanisms must ensure that differences in performance are due to differences in the mechanisms and not to differences in the application's approach to managing its workload.

8 Discussion

There is a similarity between the observations we make in this paper and observations made in work related to "receive livelock" [22] [14]. Receive livelock [22] refers to a condition where no useful progress is being made in a system because a required resource is entirely consumed by the processing of receiver interrupts. A fundamental difference between receive livelock and the problems we observe with not making enough forward progress on existing connections is that receive livelock is a result of the kernel implementation (i.e., it is caused by the kernel) while the lack of progress we observe is caused by the application itself. Therefore, the techniques used to improve performance in each case must be different.

More recently, Provos *et al.* [21] observed that although they reduced the overhead incurred in obtaining event information from the kernel, this improvement did not translate into increased server throughput. They were able to improve performance by modifying the application to observe the number of signals obtained on each call to `sigtimedwait4`. When the weighted average of the number of signals returned exceeded a threshold, they reset incoming connections instead of processing the request. While they were able to improve performance they seem less than satisfied with possible explanations for why their original improvement didn't work. From the techniques they used to improve throughput and the observations made in this paper it is likely that their original server was unable to put enough emphasis on processing existing connections under higher loads. By closing down new connections under higher loads their server was able to spend more time processing existing connections and throughput improved significantly.

Other studies [6] [24] have also reset incoming connections in order to devote more time to processing existing connections. These and other studies centered around providing differentiated or improved quality of service provide better service to some connections to the detriment of others. We consider the performance of all clients.

Welsh *et al.* [27] propose a staged event-driven architecture (SEDA) designed to simplify the implementation of well-conditioned, Internet services. Stages are connected by explicit event queues and are associated with event handlers and a thread pool. Dynamic resource controllers are deployed to automatically tune thread pool sizes, batching, and load shedding, thus controlling the resources required and applied at each stage. In our view their work is focused more on policies used to process events while our work concentrates more on understanding how Internet servers can best work in concert with existing operating system interfaces. While SEDA may be able to automatically provide the balance required to implement a high-performance server our insights can be easily applied to existing servers (two lines were added to improve the TUX server). Additionally, we believe that our work provides key insights that are essential for future comparisons of efficient event notification mechanisms.

Others researchers have looked at different aspects of what could be considered work scheduling in Internet servers. Some studies have shown [9] [11] that processing requests that require the shortest remaining processing time first can significantly improve mean response times seen by the clients. Our view is that an Internet server must continually make decisions about which work to perform next. This involves deciding not only whether to accepting one or more incoming connections or to work on any one of the large number of existing connections, but also the order in which each of these bits of work is performed. In the future we plan to expand our micro server and to use it to further investigate issues related to making these types of scheduling decisions, to study the relationships between the different types of decisions that need to be made, and to investigate operating system support that will enable the application to make better informed decisions.

Our results demonstrate that considerable care must be taken when designing and implementing high-performance web servers. In order to obtain high throughput the server must employ mechanisms that permit it to accept new connections at a sufficiently high rate. Our Accept1 server, our eager family of servers, and the CM server do not place enough emphasis on accepting new connections. As a result, their peak throughput is significantly lower than that of the AcceptInf server. However, once new connections can be accepted at a high rate the server must also ensure that a sufficient portion of time is spent making forward progress on existing connections. The AcceptInf places too much emphasis on accepting new connections. As a result, while able to obtain good peak throughput, its performance significantly degrades once the load increases past its saturation point. The AcceptInf server falls into a trap where

it accepts too many connections to work on, then it spends too much time working on too many connections before getting back to accepting new connections and by that time some of the outstanding client requests have timed out.

The Accept25 server solves this problem by limiting the number of consecutive new connections accepted. This limitation reduces the amount of work the server has to do on existing connections because there is now only a relatively small number of connections to work on. This permits the server to get back to accepting new connections quickly (before clients time-out), which is necessary for high sustained throughput.

9 Conclusions

In this paper we investigate application level Internet server design. In particular, we focus on the server's ability to dispatch network I/O events. This is done in the context of a highly parameterized micro web-server specifically implemented to permit us to study software architectures designed to avoid server meltdown during periods that require high event dispatch rates.

Our experiments show that relatively minor and seemingly sensible modifications to the server's implementation can dramatically impact the peak throughput and the throughput obtained under overload conditions. They also demonstrate that Internet server designs must carefully balance between making forward progress on existing connections and accepting new connections.

We apply these insights to another server which uses a radically different approach to dispatching events. Our experiments show that, under the workloads examined, the Linux in-kernel TUX server can be significantly improved by striking a better balance between accepting new connections and processing existing connections.

We believe that these results provide strong evidence that irrespective of the kernel event-dispatch method used, a balance must be maintained between accepting new connections, obtaining event information, and using that information to make forward progress on existing connections. When this balance is not maintained server throughput can degrade and in some cases this degradation can be substantial.

We also point out that when comparing operating system mechanisms for dispatching events, care must be taken to ensure that the servers being used are scheduling work in similar fashions. Otherwise a danger exists that conclusions will be drawn based on different event dispatching mechanisms when in fact the differences could be due to the server's approach to managing its work.

In the future we hope to integrate other event notification mechanisms into our micro-server, evaluate them, and make our server available for others to use. We plan to examine more representative workloads, techniques for multiprocessor systems, approaches to automatically and dynamically controlling event scheduling (ala SEDA [27]), and other techniques for providing kernel mechanisms that better support Internet-based server applications [18].

References

[1] M. Arlitt. *Personal communication*, September 2002.

[2] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, May/June 2000.

[3] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[4] G. Banga and J.C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998.

[5] G. Banga, J.C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[6] P. Bhoj, S Ramanathan, and S. Singhal. Web2K: Bringing QoS to web servers. Technical report, HP Laboratories, HPL-2000-61, May 2000.

[7] T. Brecht and M. Ostrowski. Exploring the performance of select-based Internet servers. Technical report, HP Labs Technical Report HPL-2001-314, November 2001.

[8] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the USENIX Annual Technical Conference*, Boston, 2001.

[9] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[10] S.L. Graham, P.B. Kessler, and M.K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPlan '82 Symposium on Compiler Construction*, pages 120–126, Boston, MA, 1982.

[11] M. Harchol-Balter, N. Bansal, B. Schroeder, and M. Agrawal. Implementation of SRPT scheduling in Web servers. Technical report, CMU, Technical Report Number CMU-CS-00-170, 2000.

[12] J. Hu, I. Pyarali, and D. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference*. IEEE, November 1997.

[13] C. Lever, M. Eriksen, and S. Molloy. An analysis of the TUX web server. Technical report, University of Michigan, CITI Technical Report 00-8, Nov. 2000.

[14] J. Mogul and K. Ramakrishnan. Eliminating receiver livelock in an interrupt-driven kernel. In *Proceedings*

of the USENIX Annual Technical Conference, pages 99–111, San Diego, CA, 1996.

[15] D. Mosberger. *Personal communication*, Sept. 2002.

[16] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67, Madison, WI, June 1998.

[17] E. Nahum, T. Barzilai, and D. Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(1), February 2002.

[18] M. Ostrowski. A mechanism for scalable event notification and delivery in Linux. Master's thesis, Department of Computer Science, University of Waterloo, November 2000.

[19] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[20] N. Provos and C. Lever. Scalable network I/O in Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2000.

[21] N. Provos, C. Lever, and S. Tweedie. Analyzing the overload behavior of a simple web server. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, October 2000.

[22] K.K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proceedings of the IEEE Global Telecommunications Conference*, pages 622–626, Orlando, FL, 1992.

[23] Red Hat, Inc. *TUX 2.1 Reference Manual*, 2001.

[24] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proceedings of the USENIX Annual Technical Conference*, Boston, June 2001.

[25] L.A. Wald and S. Schwarz. The 1999 Southern California seismic network bulletin. *Seismological Research Letters*, 71(4), July/August 2000.

[26] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles*, Banff, Oct. 2001.