Vol. 49 No. 3 IPSJ Journal Mar. 2008

Regular Paper

A Benchmark Tool for Network I/O Management Architectures

Eiji Kawai[†1] and Suguru Yamaguchi[†1]

The performance of a network server is directly influenced by its network I/O management architecture, i.e., its network I/O multiplexing mechanism. Existing benchmark tools focus on the evaluation of high-level service performance of network servers that implement specific application-layer protocols or the evaluation of low-level communication performance of network paths. However, such tools are not suitable for performance evaluation of server architectures. In this study, we developed a benchmark tool for network I/O management architectures. We implemented five representative network I/O management mechanisms as modules: multi-process, multi-thread, select, poll, and epoll. This modularised implementation enabled quantitative and fair comparisons among them. Our experimental results on Linux 2.6 revealed that the select-based and poll-based servers had no performance advantage over the others and the multi-process and multi-thread servers achieved a high performance almost equal to that of the epoll-based server.

1. Introduction

Performance requirements for network servers have been widely diversified. Server performance metrics include network bandwidth, service throughput, service time (service latency), scalability and so on. The server performance requirements largely depend on the services they serve, which have been also diversified. Today, a wide variety of network services are emerging, such as multimedia streaming services, VoIP services, large-scale scientific computing services, and global sensor network services as well as traditional e-mail and web services.

Because a network server processes multiple requests concurrently, the implementation of I/O multiplexing mechanisms, which manage I/O events to avoid I/O blockings, is fundamentally important to achieve a high performance. In this paper, we call the I/O multiplexing mechanisms *network I/O management architectures* or simply *server architectures*.

Several network I/O management architectures have been developed so far. Typical ones commonly available on Unix platforms include the multi-process/multi-thread, the I/O polling, and the event-driven architectures. However, many server developers are not well informed about the performance characteristics of each architecture and therefore they often choose an architecture relying on their intuition. This problem is caused by the difficulty in quantitative server performance evaluations from the viewpoint of server architectures and service requirements. That is, there is no simple means to evaluate and compare quantitatively the performance of the network I/O management architectures under various communication conditions.

When we evaluate the network service performance, benchmark tools are ordinarily utilized. The existing benchmark tools are categorized into two kinds: high-level benchmark tools that evaluate the server performance especially from the viewpoint of application-layer protocols and low-level ones that test the socket-level communication performance of network paths.

A high-level benchmark tool is generally developed for a specific application-layer protocol. For example, as HTTP benchmark tool, SPECweb2005[1)], Apache Benchmark (ab)[2)], httperf[3)], and http_load[4)] are popular. However, the high-level benchmark tools are not suitable to evaluate the server architectures under conditions where the communication characteristics change because their workloads are fully dependent on the application-layer protocols for which they were designed. In addition, we have to modify existing server implementations to support other server architectures, which involves a high development cost.

Low-level benchmark tools such as ttcp[5)] and netperf[6)] belong to another kind of performance evaluation tool for network services. They simply measure the data transfer performance on a single TCP/UDP socket and therefore they are not for the server performance evaluation.

The goal of this study is to develop a bench-

†1 Nara Institute of Science and Technology

mark tool that can evaluate the network I/O management architectures. Our approaches are described as follows.

- *Modularised and switchable server architectures:* We modularised the network I/O management mechanisms in our benchmark tool. This enabled a quantitative and fair evaluation of the server architectures.
- *Abstract microbenchmark:* The target of our benchmark tool is not an implementation of a specific application-layer protocol. The communication model in our tool was highly abstracted and several typical communication characteristics were parameterized. This enabled server developers to examine the server architectures even for a new service whose communication protocol had not been well-defined yet.

The rest of this paper is organized as follows. Section 2 briefly surveys the network I/O management architectures. In Section 3, we discuss typical communication characteristics and define an abstracted I/O model for our benchmark tool. Then, we describe the design and implementation of our benchmark tool in Section 4 and show the experimental results with Linux 2.6 systems in Section 5. Related work and open issues are discussed in Section 6 and Section 7. Last, we present our conclusions in Section 8.

2. Network I/O Management Architectures

Typical network I/O management architectures are the multi-process/multi-thread, the I/O polling, and the event-driven. This section describes these three architectures briefly ⋆1.

2.1 Multi-Process/Multi-Thread

A multi-process/multi-thread server allocates a process/thread to each connection. A major advantage of this architecture is implementation simplicity. Each process/thread in the server has to only treat a single connection, and I/O multiplexing is performed by process/thread switching. On the other hand, a well-known disadvantage of this architecture is process/thread management overheads. When a huge number of processes/threads are spawned in a server, it often suffers performance degradation.

2.2 I/O Polling

I/O polling is a function implemented by `select()` or `poll()` in Unix platforms. A server that utilizes the I/O polling checks the status of each connection before executing read/write I/Os, which avoids being blocked at the I/Os. An advantage of this architecture is small overheads in process/thread management because a single process/thread can treat more than one connection concurrently. However, its disadvantage is a lack of scalability. As the number of connections grows, the cost of scanning the connections also grows, which makes the service time longer 8).

2.3 Event-Driven

There are several system interfaces developed for event-driven servers. Although a standardized one is the POSIX real-time signals, many operating systems also have their own interfaces for event-driven communication processing, such as epoll 9) in Linux, kqueue 10) in BSD variants, and poll device files in Solaris. The event-driven mechanisms enable a server process/thread to handle more than one connection concurrently. In addition, they are highly scalable because the processing cost of each event does not depend on the number of the concurrent connections. On the other hand, they are not much popular compared with the other architectures for some reasons. For example, the standardized POSIX real-time signals are difficult to use with network I/Os because of their event queue overflow problem 11). Other system-dependent interfaces involve portability issues.

3. Network I/O Model

In this section, we describe the network I/O model for our benchmark tool.

3.1 Client-Server Model

Our benchmark tool adopted a typical client-server communication model. As depicted in **Fig. 1**, client hosts and a server host are connected to each other via a network switch. During tests, a large number of concurrent connections are established between the clients and the server, on which pseudo requests and responses are transferred.

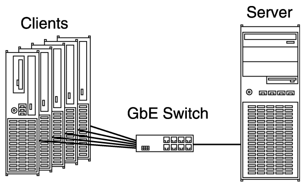


Fig. 1 The network environment for benchmark tests.

⋆1 Consult Ref. 7) for more implementation details.

The measurement target of our benchmark tool is not the performance of a specific server implementation such as the Apache web server but the performance of network I/O management architectures under a variety of communication conditions. Therefore, it consists of both a client program and a server program.

For a precise and fair performance evaluation of the server architectures, it employs a microbenchmark approach. The clients and the server do not process any data in the requests and the responses to exclude the influence of the application-layer protocol processing from the evaluation results. More specifically, the clients and the server exchange the requests and the responses which consist of random byte sequences and the verification of data receipt is done only by checking the data size.

3.2 Communication Characteristics in Benchmark Tests

Communication characteristics are the most important factors that are directly connected to the semantics of performance evaluation results. We describe the communication parameters defined in the benchmark tool and discuss the general view of their typical influence on the performance of the server architectures.

3.2.1 The Number of Connections

The number of concurrent connections a server can handle is one of the most important performance indexes. It is often regarded as synonymous with the server scalability. The server behavior when the number of concurrent connections grows is described as follows.

- *Multi-process/multi-thread:* The number of processes/threads a system can spawn is limited by various factors such as the amount of physical memory. In general, the context switching overhead and the memory consumption increase as the number of processes/threads does.
- *I/O polling:* The number of the connections the I/O polling function scans grows and thus its processing time also grows. Consequently, it causes a severe degradation of the service time.
- *Event-driven:* Processing overheads do not depend on the number of concurrent connections, which means the event-driven architecture can achieve a high scalability.

3.2.2 Throughput

Throughput has two meanings in the context of server performance: the number of requests processed per unit time at the server (request rate[⋆1]) and the amount of data exchanged between the clients and the server. When the data size distribution in the client-server communications is well-known and the server is not overloaded (i.e., all the requests that arrive at the server are processed appropriately without timeouts), we can regard the throughput to be determined by the request rate. The influence of throughput increase on the server performance can be summarized as follows.

- *Multi-process/multi-thread:* The frequency of processes/threads switchings (context switchings) increases, which raises processing overheads.
- *I/O polling:* The behavior of the processing cost is enormously complicated because it depends on the intervals of the polling I/O function invocations, the number of connections to be scanned by the I/O polling, and the frequency of network I/O events such as request arrivals at the server[12].
- *Event-driven:* Although the processing cost of each event is extremely low, the summation rises proportionately with the increase in the throughput.

3.3 Benchmark Parameters

The approach we adopted for our benchmark tool is abstract microbenchmarking, which examines the performance of the most basic server-client I/O models. Based on the discussion in this section, we defined the following configurable parameters in our benchmark.

- *The number of connections:* The total number of connections established between the clients and the server.
- *The request size:* The size of the requests sent by the clients to the server.
- *The response size:* The size of the responses sent by the server to the clients.
- *The request rate:* The number of requests sent by the clients to the server per unit time (one second).
- *The network I/O management architectures:* The server architectures described in Section 2.

4. Design and Implementation

The benchmark tool consists of a client program and a server program. We describe their design and implementation.

⋆1 Response rate may be a proper term because the request rate is not always equal to the response rate.

4.1 Client

The clients establish TCP connections to the server and send requests to the server. The number of connections and the frequency of request transmissions are given in the configuration. Each client is a multi-threaded process with two threads: a send thread and a receive thread. The processing flows in the two threads are described as follows.

Send Thread

(1) Calculate the time to send a request (the request transmission time) and wait until that time.
(2) Choose a connection randomly to send the request.
(3) Obtain a timestamp as the actual request transmission time.
(4) Write the request data into the chosen connection.
(5) Return to the first step.

Receive Thread

(1) Receive a data receipt event.
(2) Read the response data from a connection according to the event.
(3) If the whole response data is received, obtain a timestamp as the actual response reception time and record the three timestamps (the request transmission time, the actual request transmission time, the response reception time) into the log.
(4) Return to the first step.

To obtain a data receipt event in the receive thread, the epoll mechanism in Linux was adopted. At the design stage of the benchmark tool, we recognized the event-driven architecture to be the most scalable as a consequence of the general consideration described in Section 3.2. Accordingly, the current client implementation works only on Linux which implements the epoll interfaces.

Another point to be noted is the timestamps. The benchmark tool utilizes the RDTSC (ReaD Time Stamp Counter) instruction of x86 processors to obtain timestamps.

The client implementation should be done with careful considerations of time preciseness because the experimental results obtained by the benchmark tool are influenced strongly by it. As mentioned above, the preciseness of the timestamps in the benchmark tool is guaranteed by that of the CPU clocks available by the RDTSC instruction. On the other hand, the time preciseness of the client execution is achieved by the following two mechanisms.

- Request synchronization avoidance.
- Fine-grained request transmission timing control.

4.1.1 Request Synchronization Avoidance

Each client sends requests to the server at the pre-configured request rate. Thus, the request transmission timings are determined by the request rate.

Herein, if each client sends the requests at constant intervals, synchronization of the request transmissions among the clients may be possible. When two requests or more are sent from some clients in a very short term, queuing delay and processing delay are caused in the network switch and the server. These delays are not negligible especially when the request size is large. Unfortunately, the problem is that the significance of the synchronization differs in every experiment. It depends on highly delicate timings, and therefore the synchronization behavior in the experiments is not predictable even if the same configuration parameters are set in the experiments (See Appendix A.1 for a more detailed discussion).

There are two approaches to solve this problem. One is to control the request transmission timings among the clients and the other is to randomize the request transmission timings on each client. The former approach, however, is not practical when the request rate is high. For example, if the request rate is configured at 8,000 per second like in the case of the experiments in this study, the request transmission intervals are just 125 microseconds. This means each client has to synchronize its time clock with the other clients to a precision much smaller than 125 microseconds, which is almost impossible without some special time synchronization mechanisms such as GPS.

Thus, we adopted the latter approach for our benchmark tool. **Figure 2** depicts the request transmission timing randomization. The send thread of a client determines a time slot for each request from the configured request rate and randomizes the request transmission time in that time slot. Although this mechanism does not completely solve the synchronization problem, i.e., there remains some accidental synchronization by a certain ratio, it minimizes its influence on the results obtained by our benchmark tool.

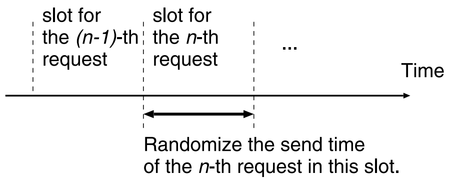

slot for the (n-1)-th request
slot for the n-th request
...
Time
Randomize the send time of the n-th request in this slot.

Fig. 2 Randomizing the request transmission timings.

4.1.2 Fine-grained Request Transmission Timing Control

As described in the previous subsection, a client has to control the request transmission timings to a precision under one millisecond. That kind of fine-grained timing control is technically difficult with the kernel timer service on Unix systems because their kernel scheduler works only to a precision of milliseconds.

In addition, execution scheduling management of the two threads (the send thread and the receive thread) is important because of the following rather contradictory reasons[⋆1].

- The scheduling delay of the receive thread should be minimized because it is equivalent to the delay of obtaining the timestamps of response receipt events. This has a great impact on the accuracy of the experimental results.
- The send thread should be scheduled as frequently as possible to achieve accurate request transmission timings.

In the implementation of the send thread of the client, we utilize a small semi-busy wait loop for the control of request transmission timings. The basic processing flow of the send thread is described in a pseudo code in **Fig. 3**. This tight main loop mechanism achieves the time preciseness of the request transmissions in a best-effort manner. Another key point in this implementation is the frequent and explicit thread switchings by invoking `sched_yield()`. This achieves the high reliability of the test results because the receive thread can obtain processor cycles as fast as possible when they are needed.

4.2 Server

The basic function of the server program is to receive requests from the clients and to send responses to them as soon as possible. The major objective of our benchmark tool is to evaluate the network I/O management architectures. To achieve this goal, we implemented our benchmark tool in a highly modularised manner as depicted in **Fig. 4**. The architectures are switchable from the configuration environment. Because the implementation codes except for the architecture modules are common in the execution path, fair performance comparisons among the architectures are possible. In the current implementation of the server, we utilize the epoll mechanism of Linux for the event-driven architecture module.

⋆1 These contradictory requirements are the major reason why real-time systems such as real-time Linux were not utilized.

```
for (;;) {
  // calculate the request
  // transmission timing
  next_send_ts = calc_next_send_ts();
  // semi-busy wait
  for (;;) {
    if (get_now_ts() >= next_send_ts)
      break;
    // switch to the receive thread
    // if possible
    sched_yield();
  }
  // send a request
  send_req();
  // avoid processing successively
  sched_yield();
}
```

Fig. 3 The main loop of the client send thread.

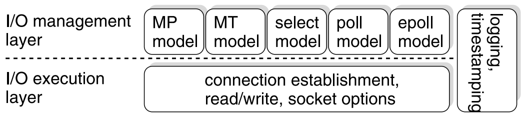


Fig. 4 The modularised server I/O management architectures.

5. Experimental Results

We conducted experiments using our benchmark tool and evaluated the performance of the server I/O management architectures on Linux 2.6. In this section, we show the experimental results and present some findings in them. We also verify the time preciseness of the request transmissions in the experiments.

5.1 Environments and Workloads

In the experiments, we prepare an environment where one server and five clients are connected via a gigabit ethernet switch. **Table 1** gives the system descriptions of the server and the clients. On these systems, we stopped all the unnecessary service processes. In addition, we changed some system configurations to support a large number of connections. First, the maximum number of open files per process and the maximum number of processes were increased. We also increased the value of

Table 1 The system descriptions of the server and the clients.

Server	CPU: Pentium III (800 MHz), MEM: 2 GB, NIC: Intel Pro/1000T 82543GC (e1000) OS: CentOS 4.4 (kernel: 2.6.9-42.0.3.EL)
Client	CPU: Pentium III (866 MHz), MEM: 512 MB, NIC: NetGear GA620T (acenic) OS: CentOS 4.4 (kernel: 2.6.9-42.0.3.EL)

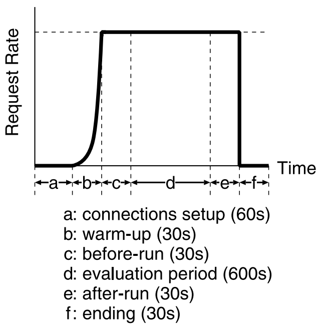


Fig. 5 The workload scenario.

FD_SETSIZE. Finally, to spawn a large number of threads, we limited the maximum stack size to 256 kbyte. This is because a large number of concurrent threads on a 32-bit system can cause virtual memory space shortage.

The benchmark tool generates network traffic between the clients and the server according to the parameters described in Section 3.3. Because a sudden full load on the server can make the system unstable and the experimental results unreliable, a gradual increase in the request rate is preferable. Thus, we implemented a workload scenario depicted in **Fig. 5**. In this workload, the request rate grows exponentially to the target for 30 seconds (the period b). The target request rate is kept for 30 seconds (the period c) to drive the system into a steady state. After that, the service time of each request is measured and recorded in the log file for 10 minutes (the period d).

5.2 Performance of Network I/O Management Architectures

In the experiments, we evaluated the server performance using two kinds of traffic pattern. One is an HTTP-like traffic model, which can be used to evaluate web server architectures. The other is a minimized traffic model, which specializes in the event processing performance of the architectures.

5.2.1 HTTP-like Traffic Model

In the HTTP-like traffic model, the parameter values described in **Table 2** were used. The performance index for the evaluations is the service time: the duration from a request transmission to a response receipt, which is the most important factor in the service satisfaction of end-users.

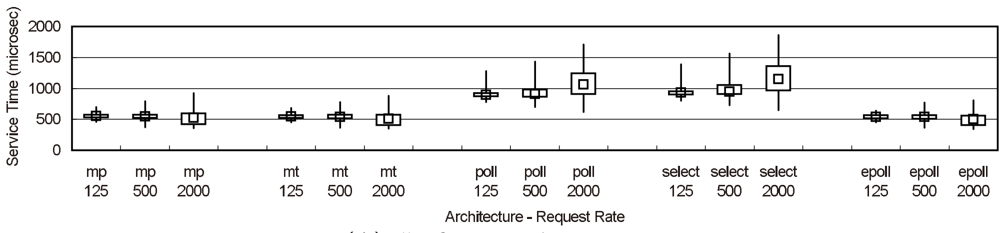
Table 2 The parameter values for the HTTP-like traffic model.

# of connections	400, 2,000, 10,000
Request size	128 byte
Response size	10 kbyte
Request rate	125, 500, 2,000
Architectures	MP, MT, select, poll, epoll

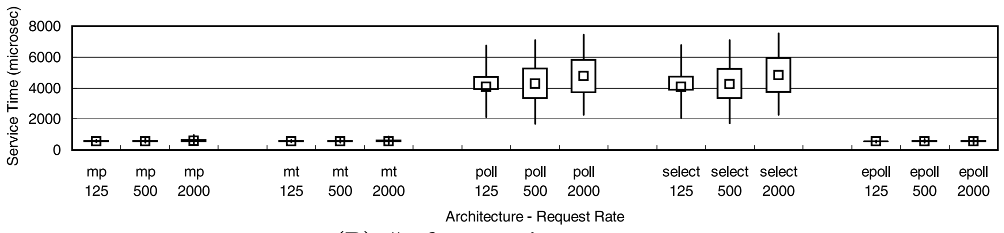
Figure 6 presents the results. The plotted service times in the graphs are 1%-tiles, 25%-tiles, median (50%-tiles), 75%-tiles, and 99%-tiles as depicted in **Fig. 7**. From the results, we can see the multi-process, the multi-thread, and the epoll servers achieved low service time. The distributions of their service times were highly stable even when the number of connections and the request rate were increased to 10,000 and 2,000 respectively. Their mean service times were approximately 500 microseconds in all tests.

On the other hand, the select and the poll servers raised their service times directly proportionately to the number of connections and the request rate. The mean service times ranged approximately from 900 microseconds to 20 milliseconds. This kind of performance issue in the select and the poll servers were also discussed in other literatures. For example, Ref. 8) showed a select-based server degraded its performance almost linearly as the number of the connections it handled concurrently increased. It also presented that a server with an event-driven approach similar to the epoll server in this study improved the service latency by an order of magnitude. The experimental results obtained in this study are consistent with those results obtained in the previous related work.

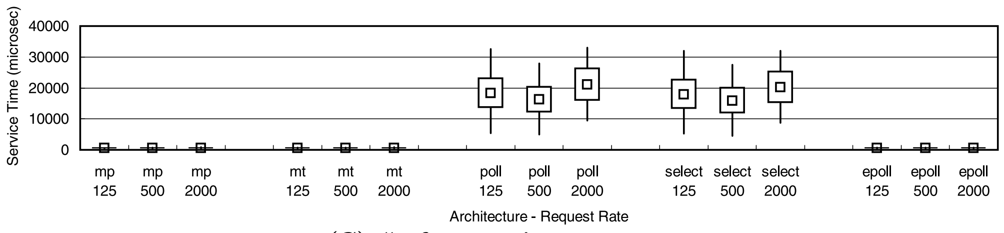
An interesting fact is that the multi-process and the multi-thread servers, which are generally recognized as unscalable architectures, achieved as low a service time as the event-driven (epoll) architecture did. At present, we have not yet conducted additional experiments utilizing other platforms such as FreeBSD and Solaris, and therefore it can be true only on Linux 2.6 that we used as the experimental platform in this study. However, we can de-



(A) # of connections = 400



(B) # of connections = 2,000



(C) # of connections = 10,000

Fig. 6 The distributions of the service times in the HTTP-like traffic model (1%-tile, 25%-tile, median, 75%-tile, and 99%-tile).

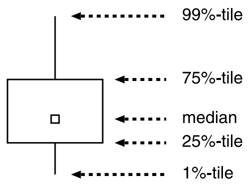


Fig. 7 The notation of the graphs.

Table 3 The parameter values for the minimized traffic model.

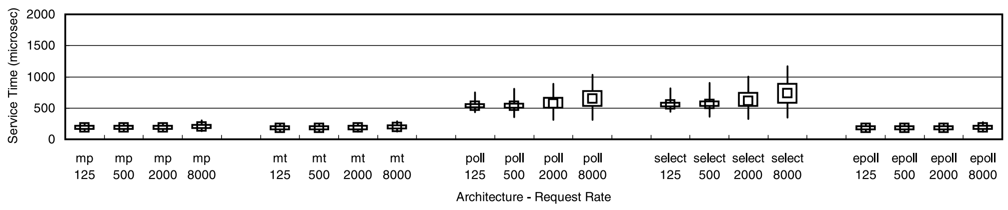
# of connections	400, 2,000, 10,000
Request size	1 byte
Response size	1 byte
Request rate	125, 500, 2,000, 8,000
Architectures	MP, MT, select, poll, epoll

rive some corollaries from it. For example, the Apache web server, which is one of the most popular web servers, has no disadvantage with its multi-process or multi-thread architecture as far as we operate it on Linux 2.6. We can also point out that it is a fruitless effort from the server architecture viewpoint to develop a new select-based, poll-based, or even epoll-based web server in order to achieve a higher scalability than that of the Apache web server.

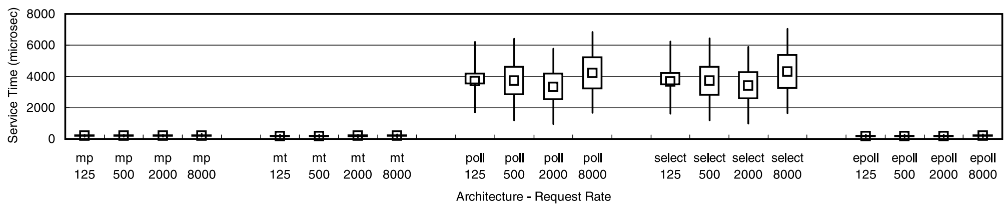
5.2.2 Minimized Traffic Model

In the minimized traffic model, we conducted experiments using the parameter values described in **Table 3**. The changes from the HTTP-like traffic model are the request size and the response size, which are both set at one byte in this model. This model minimizes the server load caused by the test traffic volume and specializes in the event processing performance evaluation. In addition, the highest request rate for the minimized traffic model was raised to 8,000 per second in contrast to the HTTP-like traffic model. The reason for which the request rate was limited to 2,000 per second for the HTTP-like traffic model was simply the server hardware performance restrictions. Actually, we conducted experiments setting the request rate at 8,000 per second for the HTTP-like traffic model, we could not obtain any valid results because the server completely got saturated.

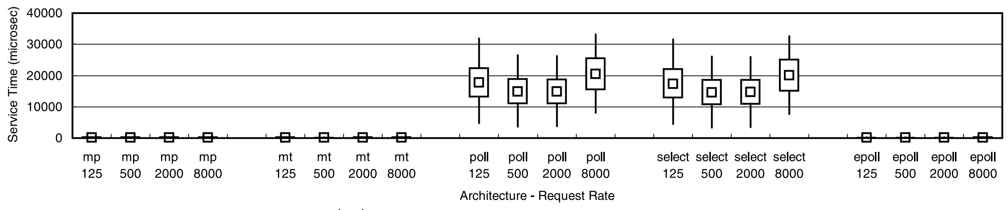
Figure 8 presents the service time distributions. The performance behavior in the minimized traffic model was similar to that in the HTTP-like model. That is, the multi-process, the multi-thread, and the epoll servers achieved



(A) # of connections = 400



(B) # of connections = 2,000



(C) # of connections = 10,000

Fig. 8 The distributions of the service times in the minimized traffic model (1%-tile, 25%-tile, median, 75%-tile, and 99%-tile).

low service times whose mean times are around 200 microseconds. The service time distributions of the select and the poll servers moved higher as the number of connections increased. Their mean service times ranged approximately between 500 microseconds and 20 milliseconds.

Compared with the HTTP-like traffic model, the mean service times of the multi-process, the multi-thread, and the epoll servers in the minimized traffic model decreased from 500 microseconds to 200 microseconds. This was mainly caused by the reduction in the data transmission time. However, the mean service times of the select and the poll servers in the minimized traffic model were almost the same as those in the HTTP-like traffic model. This is because a large portion of every service time was consumed by processing `select()` or `poll()`, and therefore the ratio of the data transmission time in the total service time was relatively small.

5.3 Preciseness of Request Transmissions

The benchmark tool implemented a control mechanism of request transmission timings. It keeps an accurate request transmission rate and avoids the synchronization of request transmissions among the clients. However, the control mechanism is based on a best-effort manner, and therefore some time lags exist between the calculated request transmission timings and the actual ones.

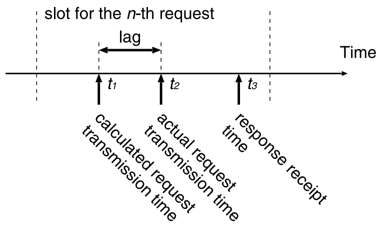
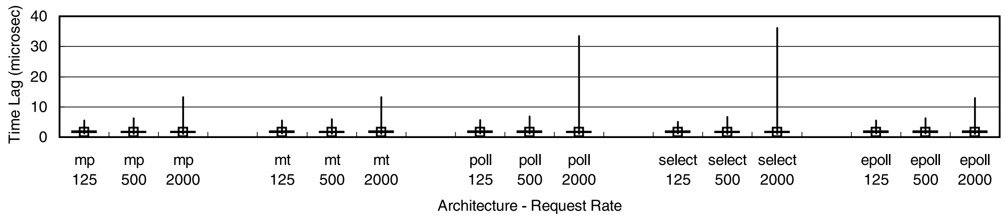

slot for the n-th request
lag
Time
t1
t2
t3
calculated request transmission time
actual request transmission time
response receipt time

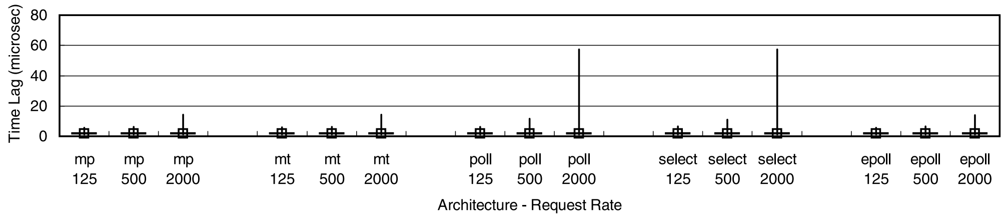
Fig. 9 Three timestamps in the log.

To investigate the time lags in the experiments, each client recorded three timestamps in the log: the calculated request transmission time (t_1), the actual request transmission time (t_2), and the response receipt time (t_3). **Figure 9** depicts the three timestamps. Herein, the time lag is defined as $t_2 - t_1$.

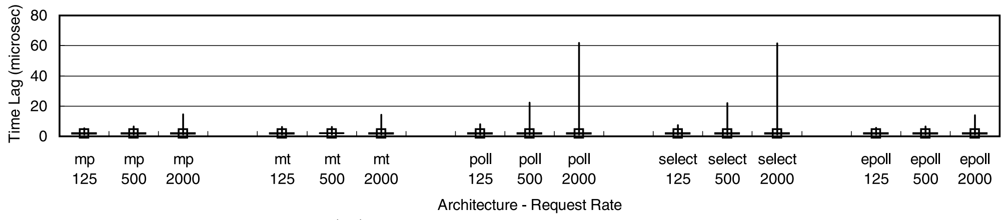
We show the distributions of the time lags in the HTTP-like traffic model and the minimized traffic model in **Fig. 10** and **Fig. 11**. From these results, we can see most of the request transmissions were started within 10 microseconds from the calculated request transmission time. Considering that the highest re-



(A) # of connections = 400

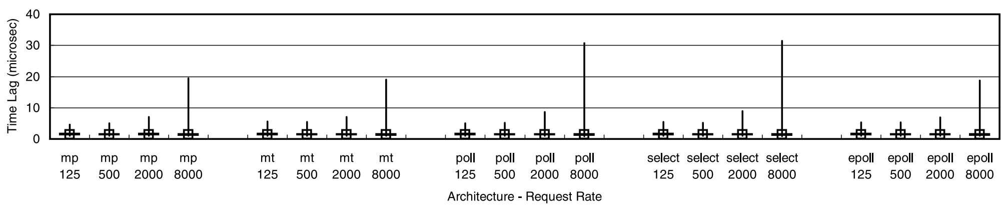


(B) # of connections = 2,000

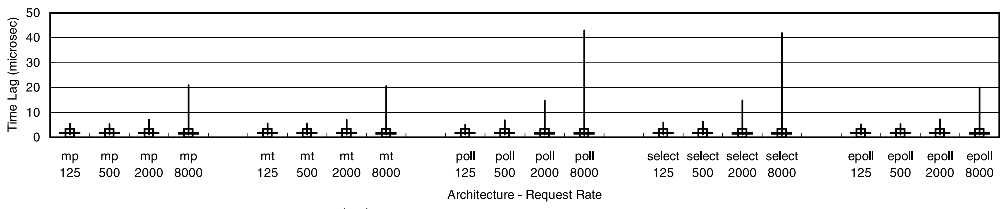


(C) # of connections = 10,000

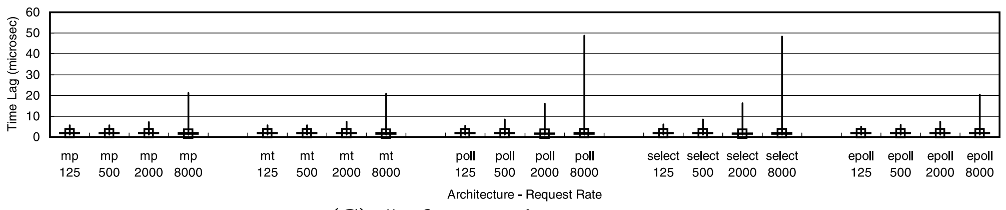
Fig. 10 The distributions of the time lags in the HTTP-like traffic model (1%-tile, 25%-tile, median, 75%-tile, and 99%-tile).



(A) # of connections = 400



(B) # of connections = 2,000



(C) # of connections = 10,000

Fig. 11 The distributions of the time lags in the minimized traffic model (1%-tile, 25%-tile, median, 75%-tile, and 99%-tile).

quest rate in the experiments was 8,000 per second, the preciseness of the request transmission control implemented in our benchmark tool is good enough.

However, we can also see that a fairly small portion of the request transmissions involved time lags larger than 10 microseconds (some 99%-tiles exceed 10 microseconds in the graphs). A possible reason to explain this is the relationship between request transmission intervals and the service time. As mentioned in Section 4.1, the client implemented the request transmission timing randomization mechanism. Therefore, when the request rate increased, the mean request transmission interval decreased and therefore the possibility of a collision between the response receipt and the next request transmission increased. This kind of collision could occur also when the service time was large compared with the request transmission intervals. Because the select and the poll servers incurred large service time especially when the number of connections was huge, some large time lags were observed with such servers even when the request rate was not large. Additionally, in the experiments with the multi-process, the multi-thread, and the epoll servers, some large time lags up to 20 microseconds were observed when the request rate was set at 8,000. In those cases, the service times (about 200 microseconds) were relatively close to the mean request transmission interval (625 microseconds[⋆1]).

6. Related Work

As mentioned in Section 1, there are many benchmark tools to evaluate the network service performance. Among them, httperf[13)] is one of the most popular benchmark tools whose technical objective is basically similar to ours. Httperf is a single-process/single-thread program and it has a request transmission control mechanism with non-blocking[⋆2] `select()` in the tight main loop. This solves the issues of possible long scheduling delays (tens of milliseconds) in operating systems. Thus, the implementation approach is almost the same as that of our benchmark tool.

On the other hand, as we revealed in this study, `select()` cannot handle a huge number of concurrent connections timely. In our benchmark tool, we solved this issue by adopting the event-driven architecture. We also improved the time preciseness of our benchmark tool by splitting the program into the send thread and the receive thread and guaranteeing that the receive thread got processor cycles quickly when some response data was received by the operating system kernel. In addition, the synchronization among the client request transmissions was avoided by randomizing the request transmission timings.

Another kind of benchmark tool related to this study is that of clustered benchmark tools. An example of such tools is Web Polygraph[14)], which is a *de facto* benchmark tool for high performance web cache/proxy products. With Web Polygraph, complicated content models and test workloads are configurable. However, it pays no attention to the synchronization issue among the clients, which is one of the contributions of this study.

Our benchmark tool implemented the most basic and popular server architectures from the practical viewpoint of server developers and operators. On the other hand, some other advanced server architectures have been proposed so far in the academic field.

The staged event-driven architecture (SEDA)[15)] is a sophisticated and promising multi-threading framework. In SEDA, service processing is divided into multiple simplified building blocks with event queues, and dynamic resource controllers allocate processing resources including threads to the stages optimizing the system performance. Because our benchmark tool focuses on the most basic network I/O model, i.e., a simple client-server data exchange model, it could not derive significant insights from experiments even if we implemented the SEDA architecture in it. However, in our study, we concluded that Linux 2.6 was highly scalable against the number of processes/threads and therefore we can mention that system operators of SEDA-based servers do not have to worry much about the number of processes/threads at each stage except for the situations where explicit resource management is required.

User-level threading is another approach to implement multi-thread programs. Capriccio[16)] is a user-level thread package for network

⋆1 The request rate was 8,000 and the request arrival rate at the server was 125 microseconds. In the experiments, we used five clients and therefore the mean request transmission interval on each client was 625 microseconds.

⋆2 The timeout value is always set at zero.

servers. Because the management costs of user-level threads are much lower than that of kernel-level ones, the scalability against the number of threads can be improved by an order of magnitude. However, it still requires event management mechanisms between the server threads and the underlying operating system kernel because the network I/Os are handled in the kernel. As a result, we can regard a user-level multi-thread server as a kind of event-driven server from the viewpoint of server architecture although their programming models are different from each other. As we examined in this study, the event management cost in an event-driven server is almost the same as the (kernel-level) thread management cost in a (kernel-level) multi-thread server. Therefore, even if the thread scalability can be largely improved by a user-level multi-threading, its benefit is rather confining from the viewpoint of the total system performance.

7. Open Issues

The major goal of this study is to design and implement a generic and abstracted benchmark tool for server architectures. Although we believe the most basic part of the goal was achieved, there still remain several advanced open issues.

First, the benchmark tool only measures the service times on pre-established TCP connections. A network server accepts new clients by opening a listening socket and processing new connection establishment requests. Therefore, it is desirable that the benchmark tool is able to model the management mechanisms of the listening socket. On the other hand, the listening socket can be handled independently of the other pre-established sockets. For example, a select/poll server can allocate a thread to the listening socket. Developing an appropriate server architecture that models the processing mechanism of the listening socket is an open issue.

Second, the benchmark tool and experiments can be extended to involve other parameters. For example, the experiments conducted in this study did not give consideration to network delays and packet losses. The peak performance of real-world network servers is often much lower than that pre-examined in experiments with LAN environments where there are neither significant network delays nor packet losses. Fortunately, adding those factors of real network behaviors into the experiments is independent of the benchmark tool implementation. A straightforward approach is to put a network emulator between the clients and the server.

Including the server processing time as server parameter is another example. One approach to implement the server processing time is to put some dummy load that consumes CPU cycles between receiving a request and sending a response. This can emulate a server like an HTTP server with CGI programs. Another possible approach is to insert a short sleep, which can emulate disk I/Os that do not consume many CPU cycles.

Last, establishing a method to verify technically the correctness of the benchmark tool behavior is a complex issue. Although we obtained various useful results in this study, the last question still remains whether the request data were really issued to the network strictly at the expected times or not. In other words, the current implementation of the benchmark tool cannot examine the duration between the time a client obtains a timestamp of an actual request transmission time and the time an ethernet frame that contains the request data appears in the network.

There are several approaches to resolve this question. One is recording timestamps in the network interface driver of the clients, which can reveal more precise behavior of the clients. Utilizing a network monitoring device is the most direct approach to investigate the network traffic in the experimental network. However, even with it, small additional delays may be inserted by packet mirroring in the switch and/or packet processing in the monitoring device. Also, recording timestamps in the server, typically in the network interface driver, is another approach, which puts some extra load on the server though.

8. Concluding Remarks

In this study, we developed a benchmark tool to evaluate the performance of three typical network I/O management architectures: the multi-process/multi-thread, the I/O polling, and the event-driven. Using the benchmark tool, server developers can compare the architectures quantitatively and choose the best one according to the service requirements.

From the experiments with Linux 2.6 systems, we saw the multi-process/multi-thread and the event-driven architectures achieved higher performance than the I/O polling architecture that incurred a long service time especially when a large number of concurrent connections were handled. Therefore, we can conclude that there are no longer any architectural reasons we should adopt the I/O polling architecture into network service programs.

Generally, the multi-process/multi-thread architecture is often considered to be unscalable compared with the other architectures. However, the results obtained in this study did not agree with that. One of the reasons the performance of the multi-process/multi-thread servers were high is the O(1) kernel scheduler was implemented in Linux 2.6. The processing cost (context switching cost) of the O(1) scheduler does not depend on the number of the processes/threads in a system, and therefore it is highly scalable against the number of processes/threads.

In addition, we observed almost no difference between the performance of the multi-process architecture and that of the multi-thread architecture. This was caused by the thread implementation in Linux 2.6 (LPTL: Linux POSIX Thread Library [17]). In Linux, threads are implemented using the `clone()` system call, and therefore the basic management mechanism of threads is similar to that of processes. The major difference between a process and a thread is their virtual memory space, i.e., each process has its own separated full virtual memory space and all the threads in a process share the virtual memory space of the process. However, because the multi-process server of our benchmark tool do not invoke `exec()` after invoking `fork()`, the larger portion of the virtual memory space is kept shared among the processes through the copy-on-write mechanism of virtual memory paging. This suppressed the performance difference between the multi-process architecture and the multi-thread architecture.

Those advanced features implemented in Linux 2.6, which we used in the experiments, may not be implemented in other platforms such as FreeBSD and Solaris. Our future work will focus on the investigation of the performance behaviors on such platforms and quantitative comparisons among them.

Acknowledgments This research was partially supported by the Ministry of Education, Science, Sports, and Culture, Grand-in-Aid for Young Scientist (B), 18700061, 2006.

References

1) SPECweb2005. http://www.spec.org/web2005/
2) Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html
3) httperf. http://www.hpl.hp.com/research/linux/httperf/
4) http_load. http://www.acme.com/software/http_load/
5) ttcp. http://ftp.arl.army.mil/~mike/ttcp.html
6) netperf. http://www.netperf.org/netperf/
7) Stevens, W.R., Fenner, B. and Rudoff, A.M.: *UNIX Network Programming — The Sockets Networking API*, Vol.1, 3rd edition, Addison-Wesley (2004).
8) Banga, G., Mogul, J.C. and Druschel, P.: A Scalable and Explicit Event Delivery Mechanism for UNIX, *Proc. USENIX Annual Technical Conference*, Monterey, California, USA (1999).
9) Gammo, L., Brecht, T., Shukla, A. and Pariag, D.: Comparing and Evaluating epoll, select, and poll Event Mechanisms, *Proc. Ottawa Linux Symposium*, Ottawa, Canada (2004).
10) Lemon, J.: Kqueue: A generic and scalable event notification facility, *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, USA (2001).
11) Chandra, A. and Mosberger, D.: Scalability of Linux Event-Dispatch Mechanisms, *Proc. 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, USA (2001).
12) Kawai, E., Kadobayashi, Y. and Yamaguchi, S.: Enhancing Network Server Performance by Controlling Execution Intervals of Polling I/O (in Japanese), *IPSJ Journal*, Vol.45, No.2, pp.391–401 (2004).
13) Mosberger, D. and Jin, T.: httperf: A Tool for Measuring Web Server Performance, *Performance Evaluation Review*, Vol.26, No.3, pp.31–37 (1998).
14) Rousskov, A. and Wessels, D.: High-performance benchmarking with Web Polygraph, *Software: Practice and Experience*, Vol.34, No.2, pp.187–211 (2004).
15) Welsh, M., Culler, D. and Brewer, E.: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services, *Proc. 18th Symposium on Operating Systems Principles*, Banff, Canada (2001).
16) von Behren, R., Condit, J., Zhou, F., Necula, G.C. and Brewer, E.: Capriccio: Scalable Threads for Internet Services, *Proc. 19th ACM Symposium on Operating Systems Principles*,

The Sagamore, Bolton Landing (Lake George), New York, USA (2003).
17) Drepper, U. and Molnar, I.: The Native POSIX Thread Library for Linux (2005). http://people.redhat.com/drepper/nptl-design.pdf

Appendix

A.1 Impact of Request Synchronization

In the experiments, five clients sent requests to a single server and measured the service time of each response. If more than one client sends a request simultaneously in a very short term, the service time will be worsen unfairly because the server can process those requests only serially and all ethernet frames for the responses are put finally into a single output queue for the server network interface. Without the request synchronization avoidance mechanism, each client sends requests at regular intervals and therefore the impact of the request synchronization differs from experiment to experiment depending on highly delicate timings.

Herein, we can roughly calculate the degree of the impact. In the HTTP-like traffic model, the response size was set at 10 kbyte. Because we utilized gigabit ethernet networks in the experiments, the transmission delay of a single response was at least 82 microseconds. So, if five requests are sent to the server almost at the same time, one of the responses suffers an extra delay of 328 microseconds caused by the request synchronization. Compared with the observed service times in the experiments with the multi-process, the multi-thread, and the epoll servers shown in Fig. 6, which range approximately from 300 to 800 microseconds, the impact is not negligible.

(Received May 25, 2007)
(Accepted December 4, 2007)
(Online version of this article can be found in the IPSJ Digital Courier, Vol.4, pp.138–150.)

Eiji Kawai is a research associate professor with Graduate School of Information Science, Nara Institute of Science and Technology (NAIST). He received the B.S. degree in mathematics from Kyoto University in 1996 and the M.E. and D.E. degrees in information systems from Nara Institute of Science and Technology in 1998 and 2001, respectively. He was an awarded researcher of PRESTO, Japan Science and Technology Agency (JST) from 2000 to 2003. His research interests cover various topics relevant to distributed computing environments, such as network protocols, operating systems, programming languages, server system design, and performance evaluation.

Suguru Yamaguchi received the M.E. and D.E. degrees in computer science from Osaka University, Osaka, Japan, in 1988 and 1991, respectively. Currently, he is a Professor with the Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan. He has been also a member of WIDE Project, since its creation in 1988, where he has been conducting research on network security system for wide area distributed computing environment. His research interests include technologies for information sharing, multimedia communication over high speed communication channels, network security and network management for the Internet.