Design, Implementation and Evaluation of Resource Management System for Internet Servers

Kazuhiro Azuma, Takuya Okamoto, Go Hasegawa and Masayuki Murata

Graduate School of Information Science and Technology, Osaka University

1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan

TEL:+81-6-6850-6863 FAX:+81-6-6850-6868

E-mail: {k-azuma, tak-okmt, hasegawa, murata}@ist.osaka-u.ac.jp

Abstract. A great deal of research has been devoted to solving the problem of network congestion posed against the context of increasing Internet traffic. However, there has been little concern regarding improvements in the performance of Internet servers such as Web/Web proxy servers in spite of the projections that the performance bottleneck will shift from networks to endhosts. In this paper, we propose a new resource management scheme for Web/Web proxy servers, which manages their resources for TCP connections, effectively and fairly. In the proposed system, we focus on the effective use of server resources by assigning dynamically the send/receive socket buffer according to the required size of each TCP connection, and terminating positively the idol persistent connection. Also, we validate the effectiveness of our proposed scheme through simulation and implementation experiments, and confirm conclusively that Web/Web proxy server throughput can be improved by 25% at maximum, and document transfer delay perceived by client hosts can be decreased by up to 90%.

Keywords: System design, Simulations, Experimentation with testbeds

1 Introduction

The rapid increase of the Internet users has been the impetus for much research into solving network congestion posed against the context of increasing network traffic. However, little work has been done in the area of improving the performance of Internet servers despite the projected shift in the performance bottleneck from

networks to endhosts. There are already hints of this scenario emerging as evidenced by the proliferation of busy Web servers on the present-day Internet that receive hundreds of document transfer requests every second during peak volume periods.

Web document transfer requests on the current Internet are done directly from Web servers to client hosts, or via Web proxy servers [1]. Needless to say, busy Web servers must have many simultaneous HTTP sessions, and server throughput is degraded when effective resource management is not considered, even with large network capacity. Web proxy servers must also accommodate a large number of TCP connections, since they are usually prepared by ISPs (Internet Service Providers) for their customers. Furthermore, proxy servers must handle both upward TCP connections (from proxy server to Web servers) and downward TCP connections (from client hosts to proxy server). Hence, the proxy server becomes a likely spot for bottlenecks to occur during Web document transfers, even when the bandwidth of the network and Web server performance are adequate. It is the contention that any effort expended to study ways to reduce document transfer time of Web documents must consider improvements in the performance of Internet servers.

In addition to the Web service, the service overlay networks in the current Internet often handle many TCP connections, which include CDNs (Content Delivery Networks) [2,3] and P2P networks [4]. In those networks, a busy server (peer) host has multiple TCP connections to other hosts in the heterogeneous environment and it behaves as a TCP sender and a TCP receiver simultaneously. Therefore, the effective management of the resources regarding TCP connections is an important issue for performance improvement of such networks.

In this paper, we first discuss several problems that arise from the handling of TCP connections at Internet servers, especially Web and Web proxy servers. One of these problems involves the send/receive socket buffers allocation for TCP connections. When a TCP connection is not assigned proper send/receive socket buffers size based on its bandwidth-delay product, the assigned socket buffer may be left unused or have insufficient capacity, which results in waste of the assigned resource and/or throughput degradation, especially on a Web or Web proxy server with many TCP connections. Another problem considered in this paper is the management of persistent TCP connections provided by HTTP/1.1 [5], which wastes resources at busy Web/Web proxy servers. When Web/Web proxy servers accommodate many persistent TCP connections without effective management schemes, their resources continue to be assigned to those connections even when they are inactive. This means that new TCP connections cannot be established since there is a shortage of server resources.

In [6], the authors pointed out other shortcomings of HTTP/1.1. They evaluated the performance of a Web server with HTTP/1.0 and HTTP/1.1 through three kinds of implementation experiments where the network

created the bottleneck, where CPU processing speed created the bottleneck, and where the disk system created the bottleneck in the system. Through experimental results, they presented that HTTP/1.1 may degrade Web server performance. This is because when memory is fully utilized by being assigned to mostly idle connections and a new HTTP session requires memory, the Web documents cached in memory are paged out, which means that subsequent requests for these documents will require disk I/O. They also proposed a new connection management method, called *early close*, which establishes a TCP connection at every Web objects, including embedded files. However, this does not provide a quintessential solution to resource management by Web servers, since it does not consider the remaining server resources. However, most of the past reported research on improving the performance of Web proxy servers has focused on cache replacement algorithms [7–9]. In [10], for example, the authors have evaluated the performance of Web proxy servers, focusing on the difference between HTTP/1.0 and HTTP/1.1 through simulation experiments, including the effect of using cookies and aborting document transfer by client hosts. However, little work has been done on resource management at the proxy server, and no effective mechanism has been proposed.

In order to overcome those problems, we propose socket buffer management scheme, which assigns the send/receive socket buffer according to the required size of each TCP connection. We also integrate two schemes for the send and receive socket buffer into a complete mechanism with considering their relationship. We further propose a connection management scheme that prevents newly arriving Web document transfer requests from being rejected at the proxy server due to lack of resources. The scheme involves the management of persistent TCP connections, which intentionally tries to dynamically close them when the server resources are shorthanded.

We verify the effectiveness of our proposed scheme through simulation and implementation experiments. In the simulations, we evaluate its basic performance and characteristics by comparing these with those of the original proxy server. Further, we discuss the results of implementation experiments, and confirm that Web proxy server throughput can be improved by 50%, and document transfer time perceived by client hosts can be decreased significantly.

The rest of this paper is organized as follows. In Section 2, we provide an outline of Internet servers, such as Web servers and Web proxy servers, and discuss the advantages and disadvantages of persistent TCP connections. In Section 3, we propose a new resource management scheme for Web/Web proxy servers, and confirm its effectiveness by detailing the results we obtained in our simulation experiments and implementation experiments in Sections 4 and 5. Finally, we present our concluding remarks in Section 6.

2 Background

In this section, we first describe the background to our research on Web servers and Web proxy servers in Subsection 2.1. We then discuss the potential that persistent connections have to improve Web document transfer times. However, as will be clarified in Subsection 2.2, it requires a careful treatment at the proxy servers. Although we focus on the Web/Web proxy servers, the discussion can be applied to the other kind of Internet services using many TCP connections such as CDNs and P2P networks.

2.1 Web/Web proxy servers

The network bandwidth of the current Internet has increased due to the previous researches, and the number of Internet users has also risen rapidly. Consequently, a Web server has to accommodate many TCP connections from Web client hosts, and especially as it receives hundreds of document transfer requests every second during peak periods. A Web proxy server has to accommodate a large number of connections from Web client hosts as well as to Web servers. Thus, even when the bandwidth of the network is efficiently large, data transfer throughput is degraded since the Web/Web proxy servers have non-optimal performance.

In the past literature, a number of studies have characterized Web server performance [11–14]. Also, some researchers have compared and evaluated the performance of Web/Web proxy servers for HTTP/1.0 and HTTP/1.1 in [6, 10, 15]. Various studies have focused on cache replacement algorithms [7–9]. However, little work has been done on the management of server resources regarding TCP connections. Server resources are finite and cannot be increased when the server is running. If the remaining resources are limit, the server cannot function fully and this results in degraded server performance.

The resources at the Web/Web proxy servers that we focus on in this paper are *mbuf*, *file descriptor*, *control blocks*, and *socket buffer*. These are closely related to the performance of TCP connections when transferring Web documents. Mbuf, file descriptor, and control blocks are resources for TCP connections. The socket buffer is used for storing transferred documents through TCP connections. When there are few resources, the Web/Web proxy servers are unable to establish a new TCP connection. Then, the client host has to wait for existing TCP connections to close and for their assigned resources to be released. If this cannot be accomplished, these servers reject the request.

In what follows, we describe the resources of Web proxy servers and how they deal with TCP connections. Although we have considered FreeBSD system [16] in our discussion, we believe that the results, in essence,

can be extended to other OSs, such as Linux.

Mbuf

Each TCP connection is assigned an *mbuf*, which is located in the kernel memory space and used to move the transmission data between the socket buffer and the network interface. When the data size exceeds the size of mbuf, the data is stored in another memory space, called the *mbuf cluster*, which is listed to the mbuf. Several mbuf clusters are used for storing data based on its size. The number of mbufs prepared by the OS is configured in building the kernel; the defaults is 4096 in FreeBSD [17]. Since each TCP connection is assigned at least one mbuf when established, the default number of connections the server can simultaneously establish is 4096. This would be too small for busy servers.

File descriptor

A *file descriptor* is assigned to each file in a file system so that the kernel and user applications can identify it. This is also associated with a TCP connection when it is established, and is called a *socket file descriptor*. The number of connections that can be established simultaneously is limited to the number of file descriptors prepared by the OS. The number of default file descriptors is 1064 in FreeBSD [17]. In contrast to mbuf, the number of file descriptors can be changed after the kernel is booted. However, since user applications, such as Squid [18], occupy memory space based on the number of available file descriptors when they are booted, it is very difficult to inform the applications of the change in the number at run time. That is, we cannot dynamically change the number of file descriptors used by the applications.

Control blocks

When establishing a new TCP connection, it is necessary to use more memory space for data structures that are used in storing connection information, such as `inpcb`, `tcpcb`, and `socket`. The `inpcb` structure is used to store source and destination IP addresses, port numbers, and other details. The `tcpcb` structure is for storing network information, such as the RTT (Round Trip Time), RTO (Retransmission Time Out), and congestion window size, which are used by TCP's congestion control mechanism [19]. The `socket` structure is used for storing information about the socket. The maximum structures that can be built in the memory space is initially 1064. Since the memory space for these data structures has been set in building the kernel and remains

5

unchangeable while the OS is running, a new TCP connection cannot be established as the amount of memory spaces is limited.

Socket buffer

The socket buffer is used for data transfer operations between user applications and the sender/receiver TCP. When the user application transmits data using TCP, the data is copied to the send socket buffer and is subsequently copied in the mbufs (or mbuf clusters). The size of the assigned socket buffer is a key issue in the effective data transfer by TCP. Suppose that a server host is sending TCP data to two client hosts; one a 64 Kbps dial-up (say, client A) and the other a 100 Mbps LAN (client B). If the server host assigns equal size send socket buffers to both client hosts, it is likely that the amount of assigned buffer will be too large for client A and too small for client B, because of the differences in the capacity (more strictly, bandwidth-delay products) of their connections. A compromise in buffer usage should be considered so that buffers can be effectively allocated to both client hosts.

2.2   Persistent TCP Connection

In recent years, many Web/Web proxy servers and client hosts have supported a *persistent connection* option, which is one of the most important functions of HTTP/1.1 [5, 15]. In the older version of HTTP (HTTP/1.0), the TCP connection between the server and client hosts is immediately closed when document transfer is completed. However, since Web document size have many in-line images, it is necessary to establish TCP connections many times to download them in HTTP/1.0. This results in a significant increase in document transfer time since the average Web documents at a typical Web servers is about 10 KBytes [11, 20]. The use of the three-way handshake at each TCP connection establishment makes the situation worse.

In HTTP/1.1 the server preserves the status of the TCP connection, including the congestion window size, RTT, RTO, and ssthresh, when it finishes document transfer. It then re-uses the connection and its status when other documents are transferred using the same HTTP session (and corresponding TCP connection). The three-way handshake can thus be avoided and latency reduced. However, the server maintains the established TCP connection, irrespective of whether the connection is active (being used for packet transfer) or not. That is, the resources at the server are wasted when the TCP connection is inactive. This results in a significant portion of resources being wasted to maintain these numerous persistent TCP connections.

One possible solution to this problem may be to simply discard HTTP/1.1 and to use HTTP/1.0, as the latter

closes the TCP connection immediately after document transfer is complete. However, as HTTP/1.1 has other elegant mechanisms, such as pipelining and content negotiation [5], we should develop an effective resource management scheme. Our solution is that as resources become limited, the server intentionally closes persistent TCP connections that are unnecessarily wasting them at the server. We will describe our scheme in detail in the next section.

3   Algorithm

In this section, we propose a new resource management scheme that is suitable for Web/Web proxy servers, which solves the problems identified in the previous section.

3.1   Socket Buffer Management Scheme

As explained previously, Web/Web proxy servers have to accommodate a numerous TCP connections. Consequently, the performance of the servers is degraded when the proper size of send/receive socket buffers are not assigned. Furthermore, when we consider Web proxy servers, resources of both of the sender and receiver sides should be taken into account. We propose a scalable socket buffer management scheme, called Scalable Socket Buffer Tuning (SSBT) in this paper, which dynamically assigns send/receive socket buffers to each TCP connection.

3.1.1   Control of Send Socket Buffer

Previous research has assumed that the bottleneck in data transfer is not in the endhost but in the network. Consequently, the allocation size of send socket buffer is fixed, for example in the previous version of FreeBSD [16], this was 16 KBytes and it is 32 KBytes in the current version. The assigned capacity is not large enough for the high-speed Internet or it is too large for narrow links. Therefore, it is necessary to assign a send socket buffer to each TCP connection with considering its bandwidth-delay product. In [21], we proposed a control scheme of the send socket buffer assigned to TCP connections and confirmed its effectiveness by simulation and implementation experiments. In what follows, we summarize the scheme briefly.

The equation-based automatic TCP buffer tuning (E-ATBT) we proposed solves the above problem. In E-ATBT, the sender host estimates the 'expected' throughput for each TCP connection by monitoring three parameters (packet loss probability, RTT, and RTO values). It then determines the required buffer size of the

connection from the estimated throughput, not from the current window size of the TCP as the ATBT scheme proposed in [22] does. The estimation method used to estimate TCP throughput is based on the analysis results obtained in [23]. The parameter set (*p, rtt* and *rto*) is obtained at the sender host as follows. *Rtt* and *rto* can be directly obtained from the sender TCP. Also, the packet loss rate *p* can be estimated from the number of successfully transmitted packets and the number of lost packets detected at the sender host via acknowledgement packets.

We denote the estimated throughput of connection i by $\overline{\rho_i}$. From $\overline{\rho_i}$, we simply determine B_i, the required buffer size of connection i, as;

$$B_i = \overline{\rho_i} \times rtt_i$$

where rtt_i is the RTT value of connection i. By this mechanism, a stable assignment of the send socket buffers to TCP connections is expected to be provided if the parameter set (*p*, *rtt*, and *rto*) used in the estimate is stable. However, in ATBT, the assignment is inherently unstable even when the three parameters are stable, since the window size oscillates more significantly regardless of the stability of the parameters.

As in ATBT, our E-ATBT also adopts a max-min fairness policy for re-assigning excess buffers. Different to the ATBT algorithm, however, E-ATBT employs a *proportional* re-assignment policy. That is, when an excess buffer is re-assigned to connections requiring more buffer, the buffer is re-assigned proportionally to the required buffer size calculated from the analysis. Whereas ATBT re-assigns excess buffers equally, since it has no means of knowing the expected throughput for the connections.

3.1.2 Control of Receive Socket Buffer

As was case for the send socket buffer, most past research has assumed that the receive socket buffer at the TCP receiver host is sufficiently large. Many current OSs assign a small, fixed-sized receive socket buffer to each TCP connection. For example, the default size of the receive socket buffer is fixed at 16 or 56 KBytes in FreeBSD systems. As reported in [24], however, this size is now regarded as small because network capacity has dramatically increased on the current Internet, and the performance of servers has also increased. Furthermore, similar to the send socket buffer for the sender TCP, the appropriate size for a receive socket buffer should be changed based on network conditions involving available bandwidth and the number of competing connections. Therefore, the receive socket buffer assigned to the TCP connection may be insufficient or remain unused when

it is not appropriately assigned.

In [25, 26], the authors proposed the control algorithm of the receive socket buffer, which is based on the monitoring results of the received data packets for every RTT. However, it needs to estimate the RTT values at the receiver side TCP, which may be quite difficult in bulk data transfer. Ideally, the receive socket buffer should be set to the same size as the congestion window size of the corresponding sender TCP, to avoid performance limitations. The problem is that the receiver cannot be informed of the congestion window size of the sender host. Furthermore, the receiver TCP does not maintain RTT and RTO values as in the sender TCP and the packet loss probability is very difficult to obtain. However, the sender TCP's congestion window size can be estimated by monitoring the utilization ratio of the receive socket buffer and the occurrence of packet losses in the network as follows.

Suppose that the data processing speed of the upper-layer application at the receiver is sufficiently high. In such a case, when the TCP packets stored in the receive socket buffer become ready to be passed to the application, the packets are immediately removed from the receive socket buffer. Let us now consider changes of the usage of the receive socket buffer, with or without packet loss in the network.

Case 1: Packet loss occurs

Here, the utilization of the receive socket buffer increases since the data packets successively arriving at the receiver host remain stored in the receive socket buffer and wait for the lost packet to be retransmitted from the sender. Assuming that the lost packets are retransmitted by the Fast Retransmit algorithm [27], it takes about one RTT for the lost packet to be retransmitted. Therefore, the number of packets stored in the receive socket buffer almost equals to the current congestion window size at the sender host, since the TCP sends data packets within the congestion window size in one RTT. Consequently, the appropriate size for the receive socket buffer should be a little larger than the maximum number of stored packets at that time. As a result, we controll the receive socket buffer as follows:

- If the utilization of the receive socket buffer becomes close to 100%, e.g. 80%, the assigned buffer size is increased since the congestion window size of the sender host is considered to be limited by the receive socket buffer, not by network congestion.

- When the maximum utilization of the receive socket buffer is substantially lower than 100%, e.g. 40%, the buffer size is decreased since excess buffer remains unused.

Case 2: No packet loss occurs

9

Different from the Case 1, the utilization of the receive socket buffer remains small. Two situations can be considered: (a) the assigned size of the receive socket buffer is sufficiently large so that the congestion window size of the sender host is not limited, and (b) the assigned size of the receive socket buffer is so small that it limits the sender's congestion window size. This depends on whether the bandwidth delay product of the TCP connection is larger than the assigned receive socket buffer size or not. Since the receiver host cannot know the bandwidth-delay product of the TCP connection, we distinguish between the two cases by increasing the assigned receive socket buffer and monitoring the change in throughput for receiving data packets from the sender as follows:

- When the throughput does not increase, it corresponds to case (a). Therefore, we do not increase the assigned size of the receive socket buffer since it is already assigned enough.

- When the throughput increases, corresponding to case (b), the proxy server continues increasing the receive socket buffer, since it is considered that increasing the socket buffer size would allow the congestion window size at the sender to be increased.

To precisely control the receive socket buffer, we should also consider the situation where the data processing speed of the receiver application is slower than the network speed. Here, the utilization of the receive socket buffer remains high even when no packet loss occurs in the network, because the data transmission rate of the sender is limited by the data processing speed of the receiver application regardless of the receive socket buffer size. That is, increasing the receive socket buffer size has little effect on the throughput of TCP connection. Therefore, the assigned size of the receive socket buffer should remain unchanged.

Based on the above considerations, we can determine the appropriate size of the receive socket buffer using the following algorithms. The receive socket buffer is resized at regular intervals. During the i th interval, the receiver monitors the maximum utilization for the receive socket buffer (U_i) and the rate at which packets are received from the sender (ρ_i). When packet loss occurs during the i th interval, the assigned receive socket buffer (B_i) is updated through the following equations:

- $B_i = \alpha \cdot U_i \cdot B_{i-1}$ when $U_i < T_u$

- $B_i = \beta \cdot B_{i-1}$ when $U_i < T_l$

The value of these parameters used in the our simulation and implementation studies is explained in the following sections. On the other hand, when no packet loss occurs during the i th interval, we use the following

equations are used to update B_i:

- $B_i = \alpha \cdot B_{i-1}$ when $U_i < T_l$ and $\rho_i \geq \rho_{i-1}$

- $B_i = \beta \cdot B_{i-1}$ when $U_i < T_l$ and $\rho_i < \rho_{i-1}$

- $B_i = B_{i-1}$ when $U_i > T_l$

3.1.3 Handling the Relation between Upward and Downward TCP Connections

A Web proxy server relays a document transfer request to a Web server for a Web client host. Thus, there is a close relation between an upward TCP connection (from the proxy server to the Web server) and a downward TCP connection (from the client host to the proxy server). That is, the difference in expected throughput for both connections should be taken into account when socket buffers are assigned to both connections. For example, when the throughput of a certain downward TCP connection is larger than that of other concurrent downward TCP connections, the larger socket buffer size should be assigned to the TCP connection using E-ATBT. However, if the throughput for the upward TCP connection corresponding to the downward TCP connection is low, the send socket buffer assigned to the downward TCP connection is not likely to be utilized fully . When this happens, the unused send socket buffer should be assigned to the other concurrent TCP connections with smaller socket buffers, improving their throughputs.

There is one problem that must be overcome in implementing the above method. Although TCP connections can be identified with the control block, called `tcpcb`, by the kernel, the relation between the upward and downward connections cannot be determined explicitly. Therefore, we need to estimate the relation by using the following algorithm. The proxy server monitors the utilization of the send socket buffer for downward TCP connections, which is assigned by the E-ATBT algorithm. When the send socket buffer is not fully utilized, it decreases the assigned buffer size, since the low utilization of the send socket buffer is considered to be caused by the low throughput of the corresponding upward TCP connection.

3.2 Connection Management Scheme

As explained in Subsection 2.2, a careful treatment of persistent TCP connections on the Web server and Web proxy server is necessary to efficiently use resources at the server that considers the amount of remaining resources. The key idea is as follows. When the Web/Web proxy server is not heavily loaded and remaining resources are sufficient, it tries to keep as many TCP connections open as possible. When resources at the
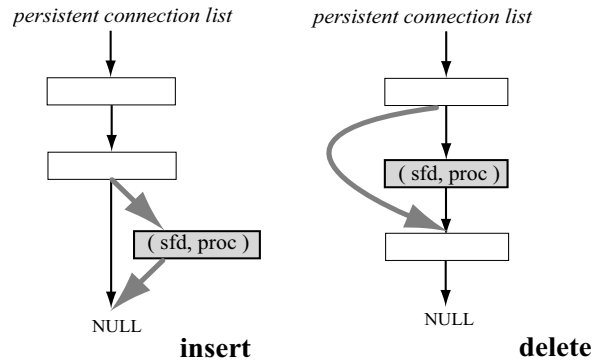
Fig. 1: Persistent connection list

server are going to be shorthanded, the server tries to close persistent TCP connections to free these resources, so that they can be used for new TCP connections.

To achieve this control the remaining resources at the Web/Web proxy server should be monitored. The resources for establishing TCP connections in our case are *mbuf*, *file descriptor*, and *control blocks*, which are resources for TCP connections. When they are limited, no additional TCP connection can be established. The amount of resources cannot be changed dynamically once the kernel and the server application are booted. However, the total and remaining amounts of resources can be monitored in the kernel system. Therefore, we introduced threshold values to utilize resources, and if one of the utilization levels for these resources reaches its threshold, the server starts closing persistent TCP connections and releases the resources assigned to those connections.

We also have to maintain persistent TCP connections at the server to maintain or close them according to how resources are being utilized. Figure 1 has our mechanism for managing persistent TCP connections at the server. When a TCP connection finishes transmitting a requested document and becomes idle, the server records the socket file descriptor and the process number as a new entry in the *persistent connection list*, which is used by the kernel to handle the persistent TCP connections. Note that new entries are added to the end of the list. When the server decides to close some persistent TCP connections, it selects connections from the top of the list. In this way, the server can close the oldest persistent connections first. When a certain persistent TCP connection in the list becomes active before being closed, or when it is closed by the expiration of the persistent timer, the server removes the corresponding entry from the list. All operations on the persistent connection list can be done with simple pointer manipulations.

To manage resources even more effectively, we add a mechanism where the amount of resources assigned to

the persistent TCP connections gradually decreases after the connection becomes inactive. No socket buffer is needed when the TCP connection is idle. Therefore, we can gradually decrease the send/receive socket buffer for persistent TCP connections by taking account of the fact that as the connection idle time continues, the possibility that the TCP connection will be terminated becomes large.

4 Simulation Experiments

In this section, we evaluate the performance of our proposed mechanism through simulation experiments using ns-2 [28]. We show the implementation overview of our proposed scheme and the results of implementation experiments in Section 5.

4.1 Simulation Settings

Figure 2 shows the simulation model. It is used to simulate a situation where many Web client hosts and Web servers in a heterogeneous environment communicate with a Web proxy server to send and receive Web documents. To do that, we intentionally set the bandwidths, propagation delays, and packet loss probabilities of the links between the proxy server and the Web clients/servers. The bandwidths of the links between the client hosts and the proxy server are changed to 100 Mbps, 1.5 Mbps, and 128 Kbps, and those between the proxy server and the Web servers are changed to 100 Mbps, 10 Mbps, and 1.5 Mbps. The packet loss probability on each link between the Web proxy servers and client hosts is selected from 0.0001, 0.001, and 0.01, and that between the proxy server and Web servers is selected from 0.001, 0.01, and 0.1. The propagation delay for each link is set to 1, 0.1, and 0.01 secs. Both of the numbers of Web servers and client hosts are fixed at 432.

In the simulation experiments, each client host randomly selects one of the Web servers and generates a document transfer request to the proxy server. The distribution for the requested document size follows that reported in [11]. That is, it is given by a combination of log-normal distribution for small documents and a Pareto distribution for large ones. The access model for the client hosts also follows that in [11], where the client host first requests the main document, and then requests some in-line images, which are included in the document after a short interval (following [11], we call it *active off time*), and then requests the next document after a somewhat longer interval (*inactive off time*). Web client hosts transfer Web document requests to the Web proxy server only when the resources at the proxy server are sufficient to accommodate that connection. When the remaining amount of server resources is shorthanded, the client hosts have to wait for other TCP

connections to be terminated and for the assigned resource to be released. After the Web proxy server accepts the request, the proxy server decides whether to transfer the requested document to the client host directly, or to download it from the original Web server and then deliver it to the client host based on the cache hit ratio h, which is fixed at 0.5 in our simulation experiments. As well as Web client hosts, the proxy server can download the requested document only when the proxy server has sufficient resources to establish a new TCP connection.

The socket buffer that the proxy server can use is divided equally between the send socket buffer and receive socket buffer. That is, when the system has 50 MBytes socket buffer, 25 MBytes are assigned equally to each buffer. We did not simulate the other limitations of the proxy server resources precisely explained in Subsection 2.1. Instead, we introduced N_{max}, the maximum number of TCP connections which can be established simultaneously at the proxy server. In the proposed scheme, we use the following parameters for the control of receive socket buffer; $\alpha = 2.0$, $\beta = 0.9$, $T_l = 0.4$ and $T_u = 0.8$.

In addition to the proposed mechanism, we conducted some simulation experiments with the traditional mechanisms, where a fixed-size send/receive socket buffer is assigned to each TCP connection for comparison. The simulation time for each experiment was 1000 sec. We compare the performance of the proposed mechanism and the traditional mechanism, focusing on the following aspects;

- **Server-side proxy throughput:** which is defined by the total transfer data size from Web servers to the proxy server divided by simulation time.

- **Client-side proxy throughput:** which is defined by the total transfer data size from the proxy server to client hosts divided by simulation time.

- **Average document transfer time:** which is defined by the average time from when client host sends Web document request to when the client host finishes receiving it.

4.2 Simulation Results

First, we evaluate the performance of the Web proxy server when the total amount of the socket buffer is changed. In this simulation, we change the total socket buffer size to 150 MBytes, 50 MBytes, 30 MBytes, and 10 MBytes, each of which is equally divided for send and receive socket buffers. N_{max} is set to 846, meaning that all the TCP connections both from Web client hosts and to Web servers are not rejected being established due to lack of other resources. Figure 3 shows the server-side proxy throughput, client-side proxy throughput,

propagation delay : 0.01 — 1 sec
loss probability : 0.001 — 0.1

h = 0.5

Web proxy server

Client Hosts
# of client hosts : 432

propagation delay : 0.01 — 1 sec
loss probability : 0.0001 — 0.01
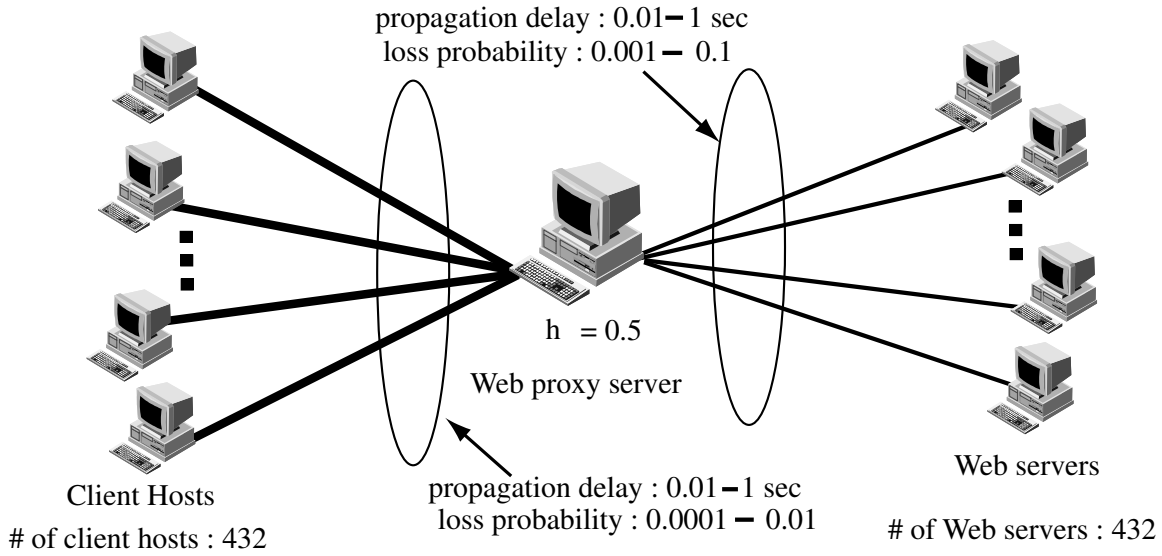
Web servers
# of Web servers : 432

Fig. 2: Network model for simulation experiments

and document transfer time. Each graph in this figure shows the results for the traditional mechanisms with various size of the send socket buffers and the receive socket buffers at the left 16 bar charts. For example, (32 KB, 64 KB) means the traditional scheme assigns a fixed 32 KBytes for the send socket buffer and a fixed 64KBytes for the receive socket buffer. The results for the proposed mechanism at the right four bar charts. Here, we did not use the connection management scheme explained in Subsection 3.2 since it shows no effect on the performance of the proposed scheme. Note that we have confirmed that the connection management scheme does not have any adverse effects in this case.

Figure 3 reveals that when the total amount of the socket buffer is small in the traditional schemes, the average proxy server throughput decreases especially when the assigned buffer size for each TCP connection is large. This is because some TCP connections do not use the assigned socket buffer when the assigned size is large. This also causes newly arriving TCP connections to wait to become established until other TCP connections have closed and released the assigned socket buffer. However, the throughput of the scheme we proposed is high even when the total amount of the socket buffer is quite small. This is because we can assign an appropriate size for the socket buffer for each TCP connection based on its estimated demand. Consequently, the excess buffers of narrow-link users are re-assigned to wide-link users who request a large socket buffer.

One possible way to improve the throughput of the proxy server in the traditional schemes is to configure the ratio of the send and receive socket buffers, instead of equally dividing send and receive socket buffer as we did in the above simulation experiments. That is, if the ratio could be changed according to the capacity
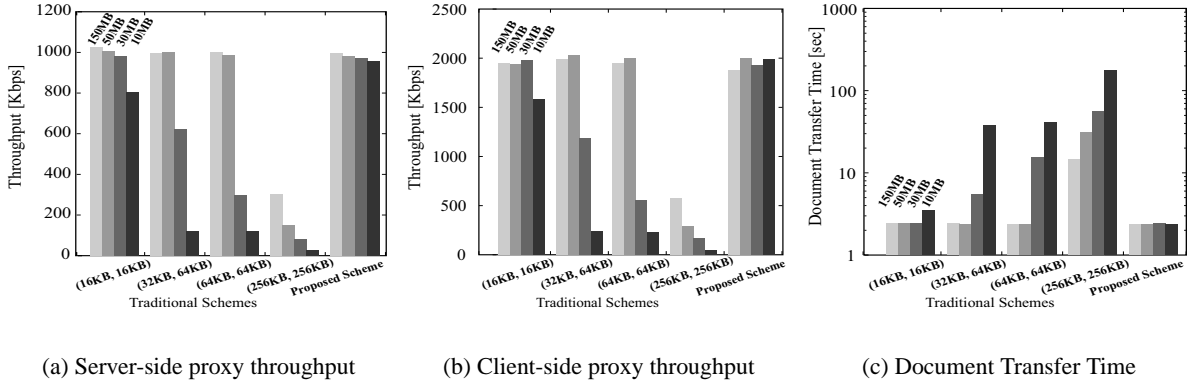
15

(a) Server-side proxy throughput   (b) Client-side proxy throughput   (c) Document Transfer Time

Fig. 3: Simulation results (1)

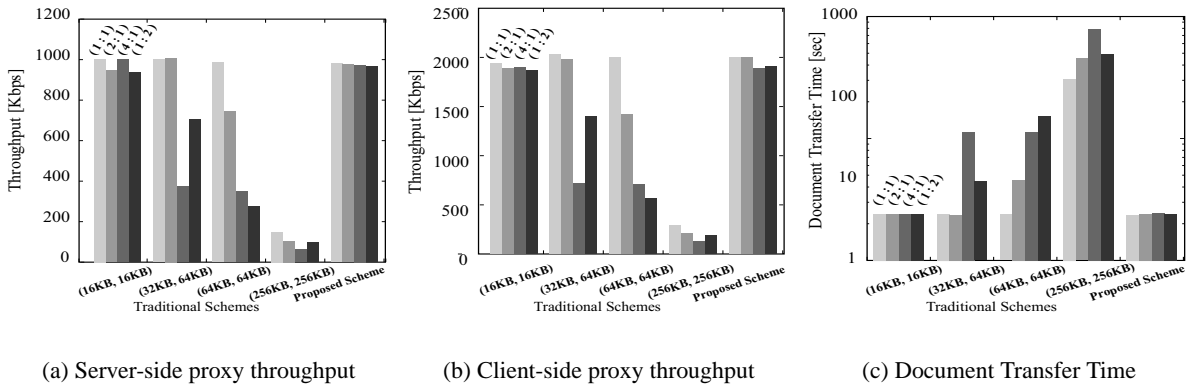(a) Server-side proxy throughput   (b) Client-side proxy throughput   (c) Document Transfer Time

Fig. 4: Simulation results (2)

required, the utilization of the socket buffer would increase, resulting in improved proxy server throughput. To investigate this further, let us look at the results when we change the ratio for the send and receive socket buffers. In this simulation, we set N_{max} to 846 and total amount of the socket buffer is fixed to 50 MBytes. We then divide the socket buffer for send and receive socket buffers in the ratios of 1:1, 2:1, 4:1, and 1:2. Figure 4 shows that in the traditional schemes, the most appropriate value of the division ratio changes when the assigned size of the socket buffer changes. It is also affected by the various factors such as the cache hit ratio, the total number of TCP connections at the proxy server, and so on. That is, it is very difficult to find the best setting of the division ratio of the send/receive socket buffer. However, proxy server throughput in our scheme still remains high even if we set the ratio incorrectly or the total socket buffer is very small. That is, we can say that our proposed scheme has good robustness against the setting of the division ratio of the socket
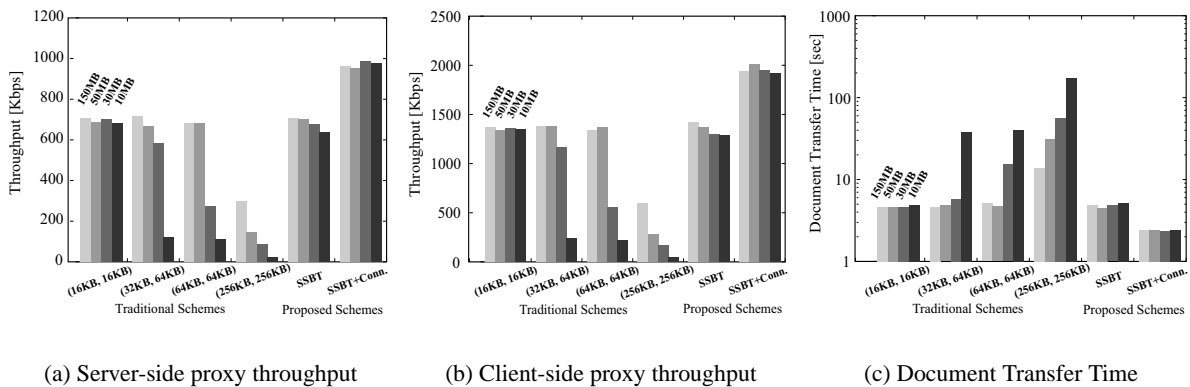
16

(a) Server-side proxy throughput   (b) Client-side proxy throughput   (c) Document Transfer Time

Fig. 5: Simulation results (3)

buffer.

Let us now discuss the results we obtain from the evaluating the connection management scheme. Here, we set N_{max} to 600, which is sufficiently small to accept all TCP connections arriving at the proxy server. The other parameters are the same as in Figure 3. Figure 5 shows the simulation results for our scheme with and without the connection management scheme (SSBT, which means when only SSBT is used, and SSBT+Conn, which means when SSBT and our proposed connection management are used). From this figure, it is obvious that in the traditional schemes, both proxy server throughput and document transfer time are worse than those in Figure 3. This small value for N_{max} means some TCP connections must wait to become established because of the lack of proxy server resources, even if most TCP connections at the proxy server are not used for data transmission and only waste the proxy server resources. This phenomena can be seen even in our scheme without the connection management scheme. On the other hand, when the connection management scheme is used, the proxy server throughput increases and document transfer time decreases. The connection management scheme proposed in this paper terminates persistent TCP connections which do not transfer any data, and the released resources are used for newly arriving TCP connections, which can start transmitting Web documents immediately. This improves the resource utilization of the proxy server, which also reduces document transfer time perceived by Web clients.

5   Implementation and Experiments

In this section, we explain the implementation overview of the proposed mechanisms. We also discuss the results we obtain in implementation experiments and confirm the effectiveness of our proposed scheme within

an actual system.

5.1 Implementation Overview

Our scheme consists of two algorithms; the socket buffer management scheme discussed in Subsection 3.1, and the connection management scheme described in Subsection 3.2. We implement them on a PC running FreeBSD 4.6, by modifying the source code of the kernel system and the Squid proxy server [18]. The total number of added source code is about 1000.

The socket buffer management scheme is composed of two mechanisms: control of send socket buffer and receive socket buffer. To control the send socket buffer, we have to obtain the 'estimated' throughput of each TCP connection established at the proxy server from the three parameters in Subsection 3.1.1. These parameters can easily be monitored at a TCP sender host, such as a Web server or a Web proxy server. We monitor these parameters in a kernel system at regular intervals. In the following experiments, we set the interval at 1 sec. We then calculate the estimated throughput of a TCP connection and assign a send socket buffer using the algorithm in Subsection 3.1.1. To control the receive socket buffer, we monitor the utilization of the receive socket buffer as described in Subsection 3.1.2. We modify the kernel system to monitor the utilization of the receive socket buffer at regular intervals, which is set to 1 sec in our implementation. Note that it should be careful to treat the receive socket buffer when the assigned buffer size is decreased in our scheme. This is because if we decrease the receive socket buffer size without sending ACK packets with a new value for the advertised window size to the TCP sender host, transferred packets may be lost due to lack of receive socket buffer. Therefore, we decrease the receive socket buffer size 0.5 sec after informing the sender host of the new advertised window size by sending an ACK packet. Also, when the advertised window size is decreased rapidly, Window Shrink problem [29, 30] may occur. However, we believe that the problem never occurs in the proposed scheme, since we decrease the receive socket buffer only when it is assigned too large to the TCP connection, based on the monitored results of the utilization of the receive socket buffer.

To implement the connection management scheme, we have to monitor the utilization of server resources and maintain an adequate number of persistent TCP connections which have been concurrently established at the proxy server, as described in Subsection 3.2. We therefore have to monitor the remaining server resources in the kernel system every second and compare them with their threshold values for the resources. Furthermore, to manage persistent TCP connections at the proxy server, we have to establish the *persistent connection list* explained in Subsection 3.2 in the kernel system.

5.2 Implementation Experiments

We next show the results of implementation experiments. Figure 6 outlines our experimental system. The Web proxy server host has dual Intel Xeon Processor 2 GHz CPUs and 2 GBytes of RAM, the Web server host has one 2 GHz Xeon CPU and 2 GBytes of RAM, and each of the five client hosts has one 1.13 GHz PentiumIII CPU and 512 MBytes of RAM. The server machines run a FreeBSD 4.6 system, and the client hosts a FreeBSD 4.8 system. The amount of proxy server resources is intentionally set such that the proxy server can accommodate up to 1000 TCP connections simultaneously. The propagation delay between the Web proxy server and the Web server is set to 300 ms and those between the Web proxy server and the client hosts are 500 ms, 150 ms, 50 ms, 10 ms and 1 ms. The threshold value, at which the proxy server begins to close persistent TCP connections in the proposed scheme, is set to 800 connections, and the parameters for the control of receive socket buffer are set as follows; $\alpha = 1.25$, $\beta = 0.75$, $T_l = 0.4$ and $T_u = 0.8$. We set the size of the proxy server cache to be 1024 KBytes, so that the cache hit ratio becomes about 0.5. The length of the persistent timer used by the proxy server is set to 15 seconds. The client hosts use httperf [31] to generate document transfer requests to the Web proxy server. It can emulate multiple users making Web accesses to the proxy server. The total number of emulated users in the experiments is set to 100, 300, 500, 700, 900, 1000, 1500 and 2000, which means that each client host emulates 20, 60, 100, 140, 180, 200, 300 and 400 users by using httperf. In the traditional system, therefore, all TCP connections cannot be accepted when the number of TCP connections at the Web proxy server is larger than 1000 due to lack of server resources. As in the simulation experiments, the access model for each user at the client hosts (the distribution for the requested document size and think time between successive requests) follows that reported in [11]. When the request is rejected by the proxy server due to lack of resources, the client host resends the request immediately. We compare the proposed scheme and the traditional scheme in which no mechanism proposed in this paper is used. Each of the experimental results below has an average value of five experiments.

Figure 7 shows the average throughput of the proxy server and the average document transfer time perceived by the each client host as a function of the number of emulated users at the client hosts. Here, we define the average throughput of the proxy server as the total size of the transferred documents from the proxy server to client hosts and those from the Web server to the proxy server divided by the experimentation time (500 sec). The document transfer time is defined as the time from when the client host sends a document transfer request to the proxy server to when the client host receives the requested document completely, which includes the

Client hosts

# of users in each Client host
20, 60, 100, 140,
180, 200, 300, 400

upper limit   : 1000 connections
threshold   : 800 connections
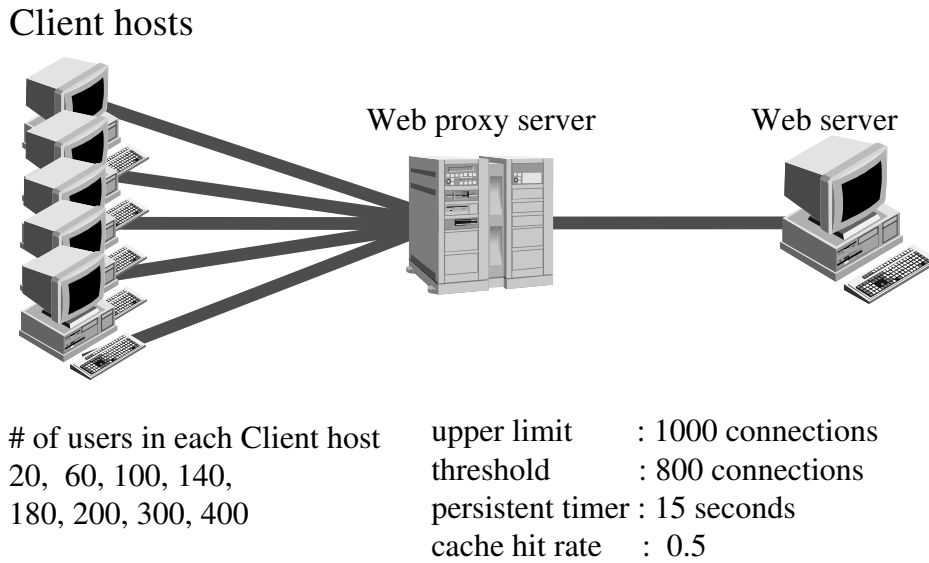persistent timer : 15 seconds
cache hit rate   : 0.5

Fig. 6: Implementation experiment system

request retransmission due to the proxy server congestion. These graphs include the 95% confidence intervals of the experimental results, which are depicted by the errorbars.

From Figure 7(a), we can observe that the traditional scheme (16 KB, 16 KB) in HTTP/1.1, which means the traditional scheme assigning a fixed 16 KBytes for the send socket buffer and a fixed 16 KBytes size for the receive socket buffer, shows the lowest throughput regardless of the number of users. This is because 16KB of the send/receive socket buffer is too small for TCP connections to provide enough throughput. Furthermore, when the number of users becomes larger than 700, the proxy server throughput degrades as the number of users increases. It means that the proxy server rejects the document transfer requests from the client hosts due to lack of server resources, even when most of the resources are wasted by the idle persistent TCP connections. Furthermore, Figure 7(b) shows that the document transfer time increases rapidly as the number of users becomes large. This is because the number of connections at the proxy server reaches 1000 and most of document transfer requests are often rejected by the proxy server.

When we increase the assigned socket buffer size to (256 KB, 256 KB) for each TCP connection, we can see that the proxy server throughput increases significantly compared with (16 KB, 16 KB) case in HTTP/1.1. This is because each TCP connection can provide enough throughput by using the large socket buffer. However, the document transfer time remains large, as shown in Figure 7(b). In this case, the proxy server assigns the very large socket buffer to each TCP connection, regardless of the condition of the connection. This results

20

(a) Average Throughput of Proxy Server
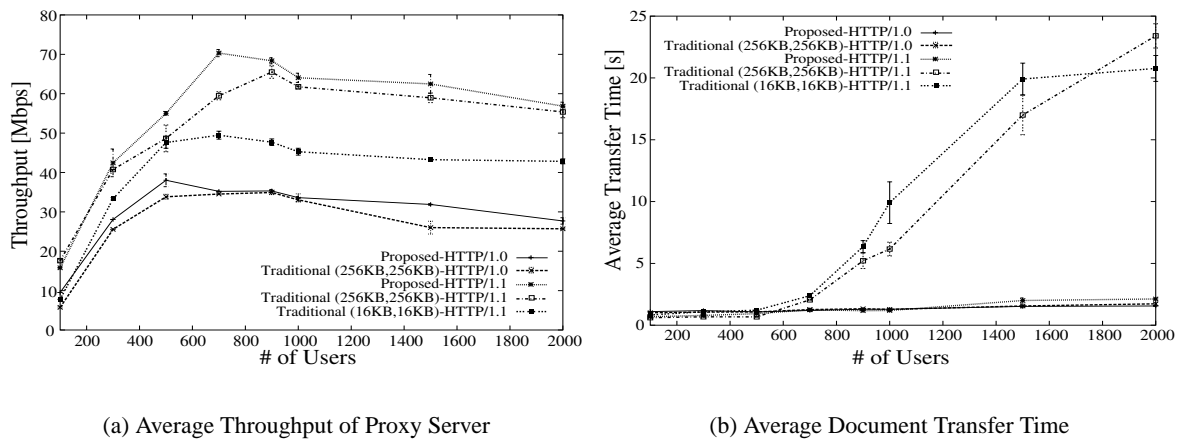
(b) Average Document Transfer Time

Fig. 7: Experimental results (1)

in lack of socket buffer for the proxy server to accept newly arriving TCP connections and the rejection of the document transfer requests.

On the other hand, our proposed scheme in HTTP/1.1 provides the highest throughput of the proxy server regardless of the number of users. This means that the proposed scheme can assign the send/receive socket buffer according to the required size by each TCP connection. Furthermore, our scheme shows quite a small document transfer time even when the number of users increases. This is because the connection management scheme successfully closes idle TCP connections wasting the resources, and accepts newly arriving TCP connections.

As we described in Subsection 2.2, one possible way to improve the performance of the traditional scheme is to discard HTTP/1.1 and use HTTP/1.0, which can avoid the persistent connection problem. From Figure 7(b), in the traditional scheme (256 KB, 256 KB) in HTTP/1.0, the document transfer time is quite smaller than the case of HTTP/1.1. However, from Figure 7(a), we can see that the proxy server throughput is much lower than the HTTP/1.1 case. This is because the overhead of three-way handshake in the continuous document transmission becomes large since the connection is terminated immediately after each document transmission. We conclude that only the proposal scheme with HTTP/1.1 which can keep the persistent connection's merit and avoid its demerit as much as possible can improve both the proxy server throughput and the document transfer time.

Furthermore, we exhibit the pretty good performance of the proposed scheme in terms of the resource utilization at the proxy server. Figure 8 shows the average size of the assigned socket buffer to all TCP con-
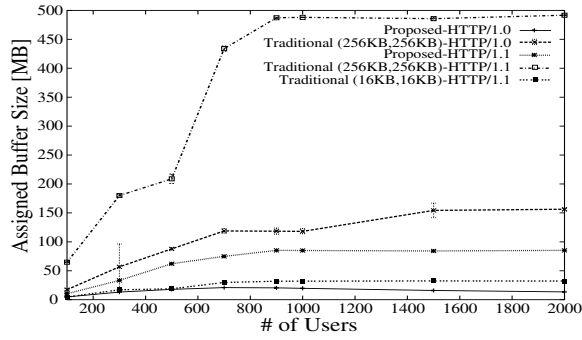
Fig. 8: Experimental results (2)

nections. It clearly shows that our proposed scheme can save the socket buffer amazingly at the proxy server. Note that the traditional scheme (256 KB, 256 KB) uses about 5 times larger socket buffer, but its throughput is lower than our proposed scheme. Also, compared with the traditional scheme (16 KB, 16 KB), our proposed scheme does not use so large socket buffer.

From the above results, we can conclude that our proposed scheme works effectively on the actual system, providing quite better performance than the traditional scheme.

6 Concluding Remarks

In this paper, we proposed a new resource management mechanism for TCP connections at Internet servers. Our proposed scheme has two algorithms. The first is a socket buffer management scheme which effectively assigns the send/receive socket buffers to heterogeneous TCP connections based on their expected throughput. It takes into account for the dependency between the upward and downward TCP connections at proxy servers. The second is a scheme for managing persistent TCP connections. It monitors server resources, and intentionally closes idle TCP connections when the remaining server resources are shorthanded. We have evaluated the our scheme through various simulation and implementation experiments, and confirmed that it can improve the proxy server performance, and reduce the document transfer time for Web client hosts.

As noted before, the application of our proposed scheme is not limited to the Internet servers. In P2P networks, for example, the maximum number of connections which can be accommodated by each peer host will increase by introducing the socket buffer control scheme proposed in this paper. Therefore, the same degree of the performance improvement shown in Section 4 and 5 can be expected.

22

References

[1] Proxy Survey, available at *http://www.delegate.org/survey/proxy.cgi*.

[2] G. Peng, "CDN: Content distribution network," Computer Science Department, Stony Brook University, Tech. Rep. Research Proficiency Exam report, Jan. 2003.

[3] K. L. Johnson, J. F. Carr, M. S. Day, and M. F. Kaashoek, "The measured performance of content distribution networks," in *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, May 2000.

[4] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, "Peer-to-Peer Computing," HP Laboratories, Tech. Rep. HPL-2002-57, Mar. 2002.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," *Request for Comments (RFC) 2068*, Jan. 1997.

[6] P. Barford and M. Crovella, "A performance evaluation of Hyper Text Transfer Protocols," in *Proceedings of ACM SIGMETRICS '99*, Oct. 1998.

[7] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proceedings of 2nd USENIX Conference on File and Storage Technologies (FAST 03)*, Mar.-Apr. 2003.

[8] L. Rizzo and L. Vicisano, "Replacement policies for a proxy cache," *IEEE/ACM Transactions on Networking*, vol. 8, no. 2, Apr. 2000, pp. 158–170.

[9] B. D. Davison, "The design and evaluation of Web prefetching and caching techniques," Ph.D. dissertation, Department of Computer Science, Rutgers University, Oct. 2002.

[10] A. Feldmann, R. Caceres, F. Douglis, G. Glass, and M. ael Rabinovich, "Performance of Web proxy caching in heterogeneous bandwidth environments," in *Proceedings of IEEE INFOCOM '99*, June 1999, pp. 107–116.

[11] P. Barford and M. Crovella, "Generating representative Web workloads for network and server performance evaluation," in *Proceedings of the 1998 ACM SIGMETRICS International Conference on Meas urement and Modeling of Computer Systems*, July 1998, pp. 151–160.

[12] R. Fonseca, V. Almeida, M. Crovella, and B. Abrahao, "On the intrinsic locality properties of Web reference streams," Boston University Computer Science Department, Tech. Rep. TR 2002-022, July 2002.

[13] M. Arlitt and C. Williamson, "Web server workload characterization: The search for invariants," in *Proceedings of the ACM SIGMETRICS '96 Conference*, Apr. 1996.

[14] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu, "The state of the art in locally distributed Web-server systems," *ACM Computing Surveys*, vol. 34, no. 2, June 2002, pp. 263–311.

[15] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol, "Key differences between HTTP/1.0 and HTTP/1.1," in *Proceedings of Eighth International World Wide Web Conference*, May 1999.

[16] FreeBSD Home Page, available at *http://www.freebsd.org/*.

[17] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System*. Reading, Massachusetts: Addison-Wesley, 1999.

[18] Squid Home Page, available at *http://www.squid-cache.org/*.

[19] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Reading, Massachusetts: Addison-Wesley, 1994.

[20] M. Nabe, M. Murata, and H. Miyahara, "Analysis and modeling of World Wide Web traffic for capacity dimensioning of Internet access lines," *Performance Evaluation*, vol. 34, no. 4, Dec. 1999, pp. 249–271.

[21] This reference is blind because of self-citation.

[22] J. Semke, J. Mahdavi, and M. Mathis, "Automatic TCP buffer tuning," in *Proceedings of ACM SIGCOMM '98*, Aug. 1998, pp. 315–323.

[23] This reference is blind because of self-citation.

[24] M. Allman, "A Web server's view of the transport layer," *ACM Computer Communication Review*, vol. 30, no. 5, Oct. 2000, pp. 10–20.

[25] M. Fisk and W. Feng, "Dynamic right-sizing: TCP flow-control adaptation," in *Proceedings of High-Performance Networking and Computing Conference (SC 2001)*, Nov. 2001.

[26] M. Gardner, W. Feng, and M. Fisk, "Dynamic right-sizing in FTP (drsFTP): An automatic technique for enhancing Grid performance," in *Proceedings of IEEE Symposium on High- Performance Distributed Computing (HPDC-11/2002)*, July 2002.

[27] W. Stevens, "TCP slow start, congestion avoidance, fastretransmit, and fast recovery algorithms," *Request for Comments (RFC) 2001*, Jan. 1997.

[28] The VINT Project, UCB/LBNL/VINT network simulator - ns (version 2), available at *http://www.isi.edu/nsnam/ns/*.

[29] J. Postel, "Transmission control protocol (TCP)," *Request for Comments (RFC) 793*, Sept. 1981.

[30] R. Braden, "Requirements for internet hosts – communication layers," *Request for Comments (RFC) 1122*, Oct. 1989.

[31] D. Mosberger and T. Jin, "httperf: A tool for measuring Web server performance," *Performance Evaluation Review*, vol. 26, no. 3, Dec. 1998, pp. 31–37.