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

Paper ID: 193 Total number of pages: 14

**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 to 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. For example, Web servers or Web proxy servers must accommodate many TCP connections simultaneously and their server throughputs degrade when resource management schemes are not considered. In this paper, we propose a new resource management scheme for those servers, which manages their resources for TCP connections, effectively and fairly. The proposed scheme has the following two mechanisms: control of send/receive socket buffers, and control of persistent TCP connections. 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 up to 50 % at maximum, and document transfer delay perceived by client hosts can be decreased by up to 30 %.

## 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 degrades when effective resource management is not considered, even with large network capacity. However, 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 this paper, we first discuss several problems that arise from the handling of TCP connections at Web or 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 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 [2], which waste 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 [3], the authors also 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 quintessential solution to resource management by Web servers, since it does not consider the remaining the server resources. However, the bulk of the past reported research on improving the performance of Web proxy servers has focused on cache replacement algorithms [4, 5]. In [6], 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 have already proposed socket buffer management scheme in [7], which assigns the send socket buffer according to the required size of each TCP connection. We have confirmed the effectiveness of the proposed scheme through some simulation and implementation experiments. In this paper we propose a control scheme of receive socket buffer of each TCP connection, and 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 the 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 up to 50 %, and document transfer time perceived by client hosts can be decreased significantly. We last note that the application of our proposed scheme is not limited to the servers, but to the Internet host handling many TCP connections simultaneously; a popular peer in P2P networks would be one example that can enjoy our proposed scheme.

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 through HTTP/1.1. 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.

### 2.1 Web/Web proxy servers

The network bandwidth of the current Internet has increased due to by the previous researches, and the number of Internet users has also risen rapidly. Therefore, 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 ([8, 9, 10]). Also, some researchers have compared and evaluated the performance of Web/Web proxy servers for HTTP/1.0 and HTTP/1.1 in [3, 6]. Various studies have focused on cache replacement algorithms [4, 5]. However, little work has been done on the management of server resources. 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. Thus, 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 4.6 [11] 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 number of defaults is 4096 in FreeBSD [12]. 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 [12]. In contrast to mbuf, the number of file descriptors can be changed after the kernel is booted. However, since user applications, such as Squid [13], 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 dynamically.

**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 [14]. 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 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 transfer of data by the 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 by HTTP/1.1

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 [2]. In the older version of HTTP (HTTP/1.0), the TCP connection between the server and client hosts is immediately closed when a document transfer is completed. However, since Web documents 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 [8, 15]. 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.

In what follows, we introduce a rough analytical estimate that can be used to determine how many TCP connections are *active* or *idle*, and explain why excessive resources are wasted in the server when native persistent TCP connections are utilized. To achieve this, we use the network topology in Figure 1, where a Web client host and a Web server are connected via a proxy server, and derive the probability that a persistent TCP connection will be *active*, i.e., in sending TCP packets. The notations in the figure, $p_c$, $rtt_c$, and $rto_c$ are the respective packet loss ratio, RTT, and RTO between the client host and the proxy server. Similarly, $p_s$, $rtt_s$, and $rto_s$ are those between the proxy and Web server. The mean throughput of the TCP connection between the proxy server and the client host, and that between the Web server and the proxy server, denoted as $\rho_c$ and $\rho_s$ respectively, can be obtained by using the analysis results presented in our previous work [16]. Note that we obtained a more accurate estimate of TCP throughput estimation than in [17] and [18] , especially for small file transfers. Using the results obtained in [16], we can derive the time for Web document transfer via a proxy server. We also introduce the parameters $h$ and $f$, which respectively represent the cache hit ratio of the document at the Web proxy server and the size of the document being transferred. Note that the proxy server is likely to cache the 'Web page', which includes the main document and some in-line images. That is, when the main document is found in the proxy server's cache, the in-line images that follow it are likely to be cached, and vice versa. Thus, $h$ is not an adequate metric when we examine the effects of persistent connections, but the following observation is also applicable to the above-mentioned case.

When the requested document is cached by the proxy server, a request to the original Web server is not required, and the document is directly delivered from the proxy server to the client host. However, when the proxy server does not have the requested document, it must be transferred from the appropriate Web server to the client host via the proxy server. Thus, document transfer time, $T(f)$, can be determined as follows;

$$T(f) = h\left(S_c + \frac{f}{\rho_c}\right) + (1-h)\left(S_c + S_s + \frac{f}{\rho_c} + \frac{f}{\rho_s}\right)$$

where $S_c$ and $S_s$ represent the connection setup times of a TCP connection between the client host and proxy server and that between the proxy server and Web server, respectively. To derive $S_c$ and $S_s$, we must consider the effect of the persistent connections provided by HTTP/1.1, which deletes the three-way handshake. Here, we define $X_c$ as the probability that the TCP connection between the client host and the proxy server can be maintained by the persistent connection, and $X_s$ as the corresponding probability that the TCP connection between the proxy server and the Web server can be maintained. Then, $S_c$ and $S_s$ can be described as follows;

$$S_c = X_c \cdot \frac{1}{2} rtt_c + (1 - X_c) \cdot \frac{3}{2} rtt_c \qquad (1)$$

$$S_s = X_s \cdot \frac{1}{2} rtt_s + (1 - X_s) \cdot \frac{3}{2} rtt_s \qquad (2)$$

$X_c$ and $X_s$ are dependent on the length of the persistent timer, $T_p$. That is, if the idle time between two successive document transfers is smaller than $T_p$, the TCP connection can be used for second document transfer. However, if the idle time is larger than $T_p$, the TCP connection has been closed and a new TCP connection must be established.

According to results in [8], where the authors modeled the access pattern of a Web client host and found that the idle time between Web document transfers follows a Pareto distribution whose probability density function is given by;

$$p(x) = \alpha k^{\alpha} x^{\alpha+1} \qquad (3)$$

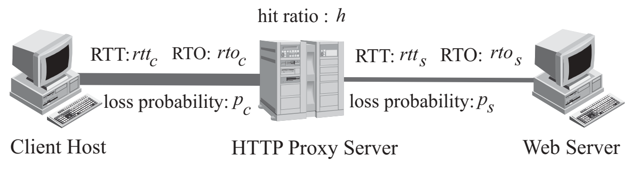


Figure 1: Analysis Model

where $\alpha$ = 1.5 and $k$ = 1. We can then calculate $X_c$ and $X_s$ as follows;

$$X_c = d(T_p) \tag{4}$$

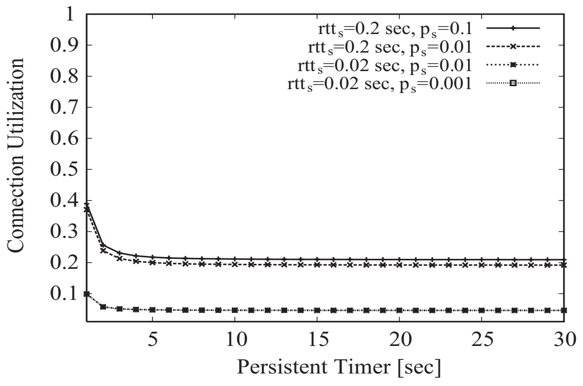
$$X_s = (1-h) \cdot d(T_p) \tag{5}$$

where $d(x)$ is the cumulative distribution function of $p(x)$. The average document transfer time, $T(f)$, can be determined from Eqs.(1)–(5). We can finally derive $U$, which is the utilization of the persistent TCP connection as follows;

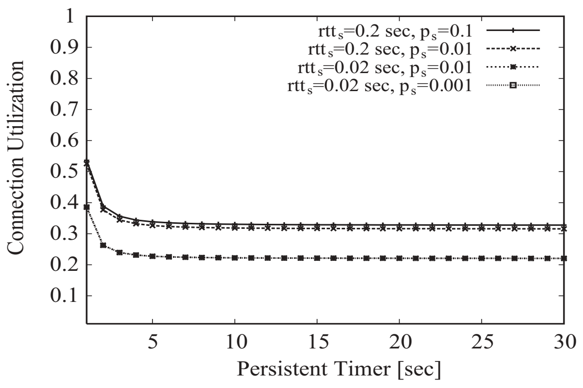
$$U = \int_0^{T_p} p(x) \cdot \frac{T(f)}{T(f)+x} dx + (1-d(T_p)) \cdot \frac{T(f)}{T(f)+T_p}$$

Figure 2 plots the probability that a TCP connection is active, as a function of the length of persistent timer $T_p$, in cases where there are various parameter sets of $rtt_c$, $p_c$, $rtt_s$, and $p_s$. Here we set $h$ to 0.5, the packet size to 1460 KBytes, and $f$ to 12 KBytes based on the average size of Web documents reported in [8]. These figures reveal that the utilization of the TCP connections is very low, regardless of the network conditions (RTTs and packet loss ratios on links between the proxy server and the client host/the Web server). Thus, if idle TCP connections are maintained at the proxy server, a large part of the resources at the proxy server are wasted. Furthermore, we can see that utilization increases when the persistent timer is short ($<$ 5 sec). This is because the smaller $T_p$ value can prevent situations from emerging where the proxy server's resources will be wasted.

One solution to this problem is 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 [2], we should develop an effective resource management scheme for it. 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.



(a) $rtt_c = 0.02$ sec, $p_s$ = 0.001



(b) $rtt_c = 0.2$ sec, $p_s$ = 0.01

Figure 2: Analysis Results: Probability that a TCP connection is active

## 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 proposed 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 [11] 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 [7], 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 in [7] 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 [19] does. The estimation method used to estimate TCP throughput is based on the analysis results obtained in [16]. 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 needing 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 [20], 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.

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. However, the sender TCP’s congestion window size can be estimated by monitoring the utilization rate 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, 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 [21], 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 controlled the receive socket buffer as follows:

- If the utilization of the receive socket buffer becomes close to 100 %, 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 %, the buffer size is decreased since excess buffer remains unused.

**Case 2: No packet loss occurs**

Different to 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 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 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 does not increase, 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 ($\rho_i$). 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 $U_i < T_l$

where $\alpha = 2.0$, $\beta = 0.9$, $T_l = 0.4$ and $T_u = 0.8$. These values are used in the simulation and implementation studies in the following sections. However, 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/Web proxy server is necessary to efficiently use resources at the server that considers the extent of remaining resources. The key idea is as follows. When the 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 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 is 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 3 has our mechanism for

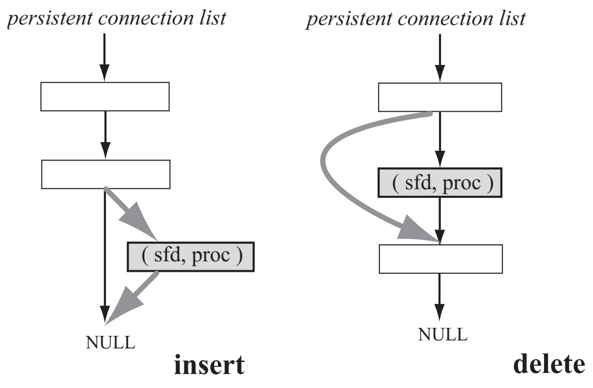


Figure 3: Connection Management Scheme

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 [22]. We show the implementation overview of our proposed scheme and the results of implementation experiments in Section 5.

## 4.1 Simulation Settings

Figure 4 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 delay, and packet loss probability 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 [8]. 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 [8], 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 [8], 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.

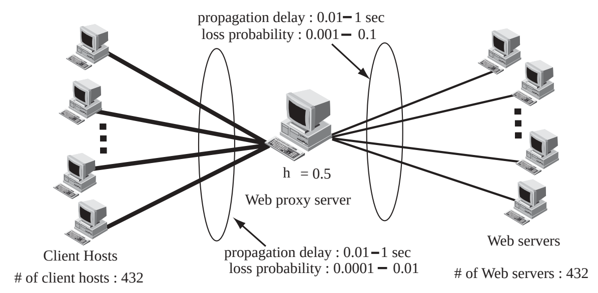

propagation delay : 0.01–1 sec
loss probability : 0.001 – 0.1
h = 0.5
Web proxy server
Client Hosts
# of client hosts : 432
propagation delay : 0.01–1 sec
loss probability : 0.0001 – 0.01
Web servers
# of Web servers : 432

Figure 4: Network Model for Simulation Experiments

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 is 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 5 shows the server-side proxy throughput, client-side proxy throughput, 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 schemes 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 4 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 5 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 for 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 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 divided the socket buffer for send and receive socket buffers in the ratios of 1:1, 2:1, 4:1, and 1:2. Figure 6 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 be 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 buffer.

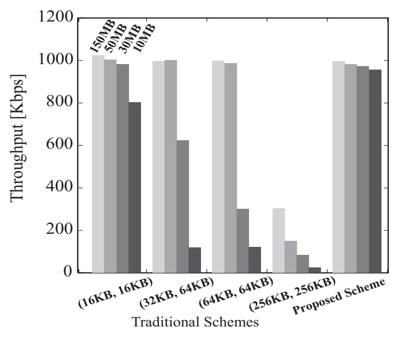
Let us now discuss the results we obtain from the evaluating the connection management scheme. Here, we set $N_{max}$ to 600, which is sufficiently small to accept all TCP connections arriving at the proxy server. The other parameters are the same as in Fig. 5. Figure 7 shows the simulation results for our scheme with and without the connection management scheme (SSBT and SSBT+Conn, respectively). From this figure, it is obvious that in the traditional schemes, both proxy server throughput and document transfer time are worse than those in Fig. 5. 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 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 of the resource utilization of the proxy server, which also reduces of document transfer time perceived by Web clients.

## 5 Implementation and Experiments

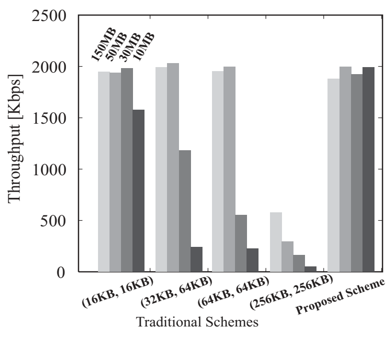
In this section, we discuss the results we obtain in implementation experiments and confirm the effectiveness of our proposed scheme within an actual system.

### 5.1 Implementation Overview

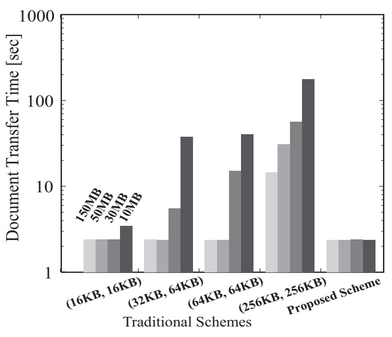
Our scheme consists of two algorithms; the socket buffer management scheme discussed in Subsection 3.1, and the connection management scheme described in Subsection 3.2. We implement it on a PC running FreeBSD 4.6,



(a) Server-side proxy throughput

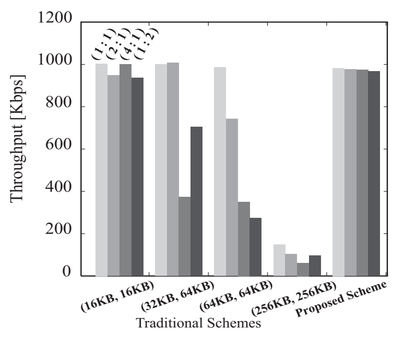


(b) Client-side proxy throughput

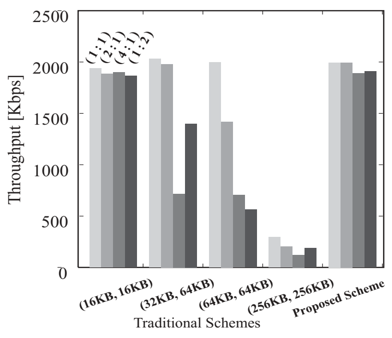


(c) Document Transfer Time

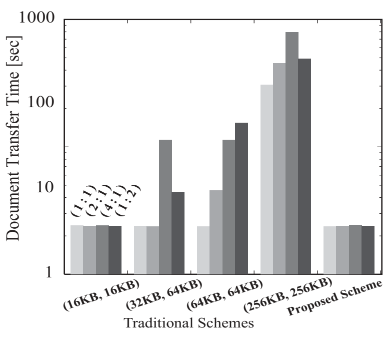
Figure 5: Simulation Results (1)



(a) Server-side proxy throughput



(b) Client-side proxy throughput



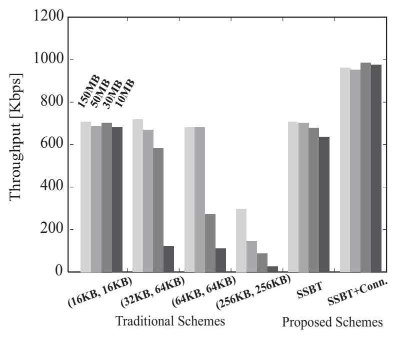
(c) Document Transfer Time

Figure 6: Simulation Results (2)

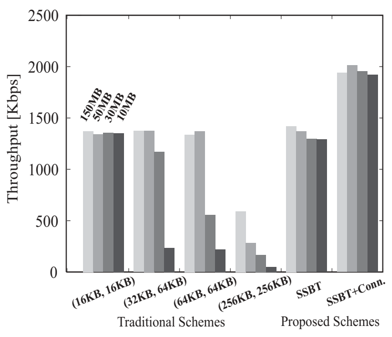
by modifying the source code of the kernel system and the Squid proxy server [13]. The total number of lines of added source code is about 1000.

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 a 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.

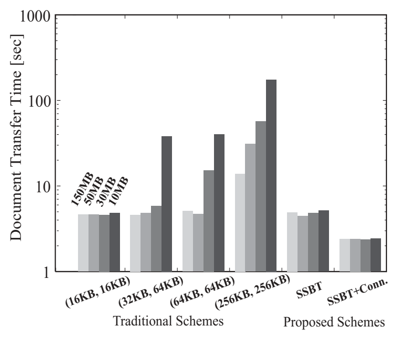
To implement the connection management scheme, we have to monitor the utilization of server resources and maintain an adequate number of persistent TCP connections which have been concurrently established at the proxy server, as described in Subsection 3.2. We therefore have to monitor the remaining server resources in the kernel system



(a) Server-side proxy throughput



(b) Client-side proxy throughput



(c) Document Transfer Time

Figure 7: Simulation Results (3)

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 8 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 2 GHz CPU and 2 GBytes of RAM, and the client host has 2 GHz CPU and 1 GByte of RAM. All of three machines run a FreeBSD 4.6 system. The amount of proxy server resources is set such that the proxy server can accommodate up to 400 TCP connections simultaneously. The threshold value, at which the proxy server begins to close persistent TCP connections, is set to 300 connections. We intentionally 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 host uses httperf [23] to generate document transfer requests to the Web proxy server, which can emulate numerous users making Web accesses to the proxy server. The number of emulated users in the experiments is set to 200 and 600. In the traditional system, therefore, all TCP connections at the Web proxy server can be established when there are 200 TCP connections, but all TCP connections cannot be accepted when this number is 600 due to the lack of server resources. As in the simulation experiments, the access model for each user at the client host (the distribution for the re-

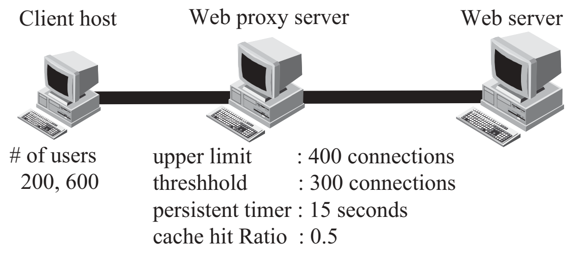
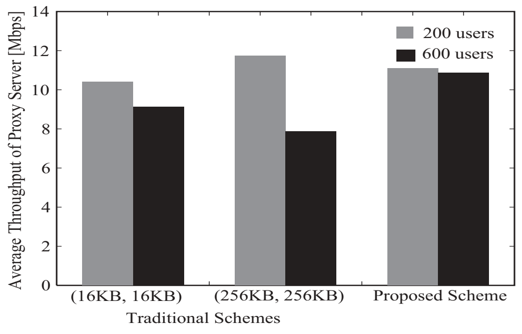

Client host
Web proxy server
Web server
# of users
200, 600
upper limit : 400 connections
threshhold : 300 connections
persistent timer : 15 seconds
cache hit Ratio : 0.5

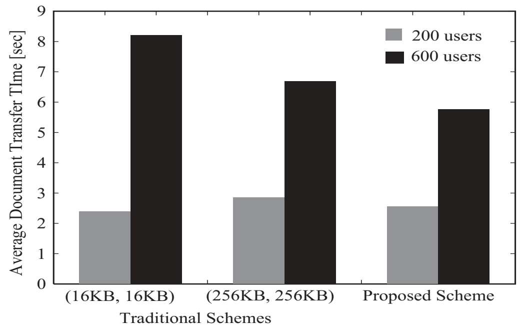
Figure 8: Implementation Experiment System

quested document size and think time between successive requests) follows that reported in [8]. 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 original scheme in which no mechanism proposed in this paper is used.

Figure 9 shows the average throughput of the proxy server, and the average document transfer time perceived by the client host. 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 divided by the experimentation time (500 sec). From this figure, when the number of users is 200, the average throughput of traditional scheme (16 KB, 16 KB) is lower than that of another traditional scheme (256 KB, 256 KB). This is because the assigned size of send/receive socket buffer is too small in spite of that the network capacity and the Web server performance is sufficiently. When the number of users increases to 600, however, the performance of the traditional schemes decreases in both terms of the proxy server throughput and the document transfer time, Especially, the traditional scheme (256 KB, 256 KB) degrades its perfor-



(a) Average Throughput of Proxy Server



(b) Average Document Transfer Time

Figure 9: Experimental Results (1)

mance significantly. This is because the proxy server resources are much shorthanded in this case, since too large size of socket buffer is assigned to TCP connections.

When we look at the results of our proposed scheme, we can observe that when the number of users is 200, the proxy server throughput is almost the same as that of the traditional scheme (256 KB, 256 KB). In 600 users case, however, our proposed scheme does not degrade its throughput in spite of that the number of users is much larger than the capacity of the proxy server. It also provides the smallest document transfer time for the Web client host. This results mean that our scheme can effectively control the persistent connections at the proxy server.

Further more, we exhibit the pretty good performance of the proposed scheme in terms of the resource utilization at the proxy server. Figure 10 shows the average size of the assigned socket buffer to all TCP connections. It clearly shows that our scheme can save the socket buffer amazingly at the proxy server. Note that the traditional scheme (256 KB, 256 KB) uses about 10 times larger socket buffer but it can provide only 75 % throughput and larger document transfer time, compared with the proposed scheme. 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.

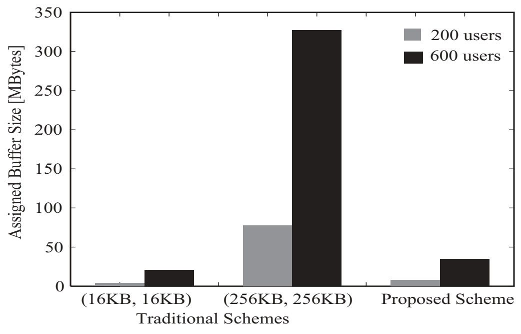


Figure 10: Experimental Results (2)

## 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.

## References

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

[2] 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.

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

[4] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, pp. 193–206, Dec. 1997.

[5] L. Rizzo and L. Vicisano, "Replacement policies for a proxy cache," *IEEE/ACM Transactions On Networking*, vol. 8, pp. 158–170, Apr. 2000.

[6] 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*, pp. 107–116, June 1999.

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

[8] 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*, pp. 151–160, July 1998.

[9] V. Almedia, A. Bestavros, M. Crovella, and A. de Oliveria, "Characterizing reference locality in the WWW," in *Proceedings of 1996 International Conference on Parallel and Distr ibuted Information Systems (PDIS '96)*, pp. 92–103, Dec. 1996.

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

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

[12] 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.

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

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

[15] 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, pp. 249–271, Dec. 1999.

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

[17] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe, "Modeling TCP throughput: A simple model and its empirical validation," in *Proceedings of ACM SIGCOMM '98*, pp. 303–314, Aug. 1998.

[18] N. Cardwell, S. Savage, and T. Anderson, "Modeling TCP latency," in *Proceedings of IEEE INFOCOM 2000*, pp. 1742–1751, Mar. 2000.

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

[20] M. Allman, "A Web server's view of the transport layer," *ACM Computer Communication Review*, vol. 30, pp. 10–20, Oct. 2000.

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

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

[23] D. Mosberger and T. Jin, "httperf: A tool for measuring Web server performance," *Performance Evaluation Review*, vol. 26, pp. 31–37, Dec. 1998.