A Dynamic Approach for Efficient TCP Buffer Allocation[*]

Amit Cohen and Reuven Cohen
Dept. of Computer Science
Technion
Haifa 32000, Israel

Abstract

The paper proposes local and global optimization schemes for efficient TCP buffer allocation in an HTTP server. The proposed local optimization scheme dynamically adjusts the TCP send-buffer size to the connection and server characteristics. The global optimization scheme divides a certain amount of buffer space among all active TCP connections. These schemes are of increasing importance due to the large scale of TCP connection characteristics. The schemes are compared to the static allocation policy employed by a typical HTTP server, and shown to achieve considerable improvement to server performance and better utilization of its resources. The schemes require only minor code changes and only at the server.

Keywords: HTTP, server performance, TCP send-buffer.

---

[*] An early version of this paper was presented in IC3N'98, The 7'th International Conference on Computer Communications and Networks, Lafayette (Louisiana), October 1998.

1 Introduction

HTTP requests are the most popular way to retrieve information over the Internet. An HTTP transaction consists of one or more TCP connections that are established between a client and a server. The performance of an HTTP server depends, to large extent, on the availability and usage efficiency of resources like bandwidth, CPU and memory. An important and extensively discussed issue is how to tune an HTTP server to use its resources efficiently, in order to achieve maximum performance.

In a typical HTTP server, the major components of the main memory are the operating system kernel, the server processes and a file cache. An important part of the kernel space is dedicated to network buffers, that are mainly employed by the server as TCP send-buffers. In a common HTTP session, the server copies the requested data to the TCP send-buffer. From there, the data is forwarded to the client by the output routines of the TCP stack. When the requested data is larger than the send-buffer size, this procedure is repeated until the whole transfer is completed.

A typical HTTP server can handle hundreds of HTTP requests simultaneously. Since each HTTP session runs over one or more dedicated TCP connections, and each connection uses a separate TCP send-buffer, all active connections must somehow share the limited amount of main memory reserved for TCP buffers. These buffers cannot be stored on a secondary storage, even if they were not a part of the kernel space. This is because of the excessive latency penalty caused by the access to the secondary storage.

This paper addresses the issue of main memory allocation to the send-buffers of active TCP connections in an HTTP server or proxy server. In particular, it concentrates upon the following two issues:

1. Local optimization: determining the optimal send-buffer size of an active TCP connection.
2. Global optimization: determining a strategy for dividing a certain amount of buffer space among a certain number of active TCP connections.

If the send-buffer of a TCP connection is not big enough, the available bandwidth of the connection cannot be fully utilized. On the other hand, a too big send-buffer would waste a resource that might have been more efficiently used by another connection. We define the optimal size of a send-buffer for a TCP connection as the smallest size that enables the connection to use the maximum available bandwidth. This paper proposes a method for local optimization that dynamically allocates send-buffers to TCP connections.

An important property of the proposed method is that it dynamically adjusts the TCP send-buffer size to the connection and server characteristics. The Internet introduces a great scale of end-users, connected to the network by means of many types of equipment: slow analogue modems, ADSL and satellite links, and high-speed routers. While a connection over an analogue modem might have a bandwidth-delay products of 1-2 segments, a connection over a satellite link might experience a bandwidth-delay product of more than 100 segments. Our method recognizes the amount of resources needed by each connection and therefore leads to a better utilization of the server resources

In a typical HTTP server, such as `Apache` [1] , all active TCP connections have an equal (configurable) bound on the send-buffer space they can use. This naive allocation policy is similar to giving each connection an equal share of the buffer-space. In the context of "global optimization", we present more sophisticated allocation policies that are based on the local optimization scheme. These policies are shown to improve the performance of a loaded web server, when memory is a limiting factor.

The rest of the paper is organized as follows. Section 2 presents an overview of the TCP send-buffer concept. Section 3 discusses the local optimization, and proposes a scheme for dynamic allocation of send-buffers. Section 4 introduces new allocation policies for global optimization, and compares them to the current naive approach. Although TCP and HTTP were heavily researched over the last few years, the TCP buffer mechanism was left fairly untouched. However, in parallel to our research, an independent research was conducted by the PSC networking group. The two researches were published simultaneously (see [4] and [7]). Section 5 discusses the similarity and the differences of the two solutions. Section 6 concludes the paper, and in Appendix A we address the issue of Round-Trip Time (RTT) measurement.

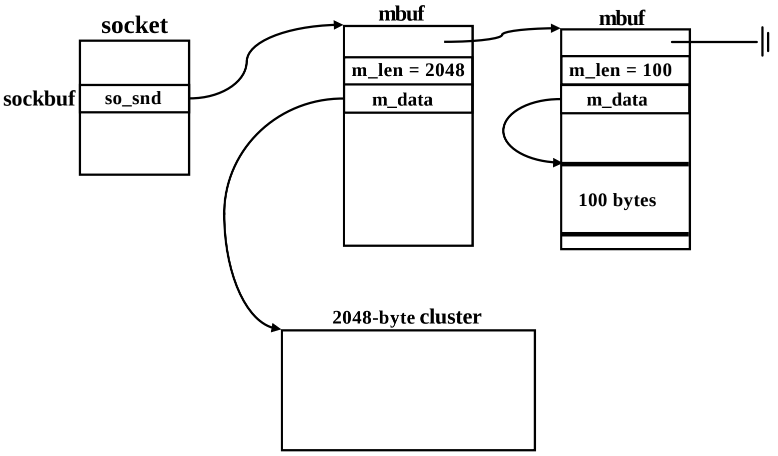

Figure 1: A TCP send-buffer.

2 TCP Send-Buffer Management

This work addresses the TCP buffer management scheme of the BSD UNIX, in particular the `4.4BSD-Lite` implementation. Since this scheme was adopted by many commercial operating systems, such as SUN Solaris, SGI IRIX and even WindRiver's real-time operating system VxWorks, the following discussion can be viewed as a model of TCP buffer management in a general HTTP server.

In `4.4BSD-Lite` (e.g [13]), a `socket` structure consists of two `sockbuf` sub-structures: one for the send-buffer (`so_snd`) and another for the receive-buffer (`so_rcv`). As shown in Figure 1, each `sockbuf` structure contains a pointer to a linked chain of `mbuf` structures. An `mbuf` structure consists of 128 bytes. Some of these bytes are used for control data, such as a pointer to the next `mbuf` in the chain, a pointer to a data buffer, a number of valid data bytes in the data buffer, and so on. The remaining bytes, up to 108, can be used for storing data. Each `mbuf` in the chain that does not contain data, points to a bigger data buffer, called `cluster`. A `cluster`, also known as `extern-buffer` or `mapped page`, is a storage unit of 1, 2 or 4 Kbytes, depending on the operating system version. Figure 1 shows a TCP send-buffer containing 2148 bytes of data, that are stored in one 2048-byte `cluster` and one `mbuf`. Note that in a TCP send-buffer the data is stored as a stream of bytes, with no packet borders. When new acknowledgments (ACKs) are received and *all* the data contained in an `mbuf` or a `cluster` is acknowledged, the buffer is deleted from the chain and appended to the free buffer list. The maximum number of `cluster`s that can be allocated is defined at system configuration.

The amount of data in the send-buffer is regulated by the following three `sockbuf` data members:

1. `sb_cc` indicates the current number of unacknowledged data bytes in the entire send-buffer.
2. `sb_hiwat` (high water mark) is an upper bound on `sb_cc`; thus, it indicates the maximal number of data bytes the send-buffer can contain.
3. `sb_lowat` (low water mark) is a lower bound on the amount of free space (`sb_hiwat-sb_cc`) the send-buffer should have before it can accept more data from the application. A server process that tries to add new data into the send-buffer when the amount of free space is below `sb_lowat` is suspended. When a new ACK is received, a process that was suspended due to lack of free space in the send-buffer is resumed. However, such a process is immediately re-suspended if the released space is not large enough.

In most UNIX HTTP servers the values of `sb_lowat` and `sb_hiwat` are set during the server configuration. Therefore, every TCP connection has the same bound on the send-buffer. When the size of an HTTP response is much bigger than `sb_hiwat`, such an approach is equivalent to allocating an equal portion of the buffer pool to every TCP connection. As shown later, this naive policy has a bad impact on server performance.

Procedure 2.1 presents a pseudo-code of a WWW server sending an HTTP response to a client:

Procedure 2.1

```
while (nbytes=fread(requested-file, array, size-of-read,...)) {
    ...
    send(socket,..., array, nbytes);
    ...
}
```

During every iteration, a block of a few Kbytes is read from the requested file. Then, system call `send` is called in order to transfer these bytes into the TCP send-buffer. The socket layer function `sosend` is invoked as a result of the call to `send`. A pseudo code describing the operation of `sosend` is presented in procedure 2.2.

Procedure 2.2

```
repeat {
    while (sb_hiwat-sb_cc<sb_lowat) wait;
    fill one network buffer (mbuf or cluster);
    call TCP output procedure;
} until all data in the block has been transferred to the send-buffer;
```

First, as explained before, the process waits until the send-buffer has sufficient empty space. Then, a block of data is copied into a newly allocated buffer which is afterwards appended to the send-buffer. When TCP flow control scheme, to be discussed in the next section, allows to send a new segment, a new segment with an appropriate TCP header is formed by the TCP output procedure. Then, the IP output routine is called and the segment is enqueued for transmission on the network interface. Function `sosend` returns after all the received data is transferred to the send-buffer. A more detailed description of the TCP buffering mechanism is given in [4].

Throughout the paper we consider a server whose internal structure is similar to what have been described so far. In particular, it is assumed that the size of the TCP send-buffer is regulated by high and low water marks, and that the send-buffer is constructed from basic storage units.

3 Local Optimization of Buffer Allocation

This section proposes a method for approximating the optimal send-buffer size for every TCP connection. As already indicated, the optimal size is the minimum that enables the connection to work at maximum speed. In the context of local optimization we consider the performance of a single connection. Section 4 expands the discussion by considering the case where many connections exist, and global optimization is sought.

3.1 TCP Congestion Control

We start with a short overview of TCP congestion control mechanism. This mechanism dictates the actual rate data is transmitted by a TCP connection into the network. Hence, it directly affects the optimal size of the send-buffer.

A TCP connection is bi-directional in the sense that both sides can send data. However, since this paper concentrates on the data transmitted by an HTTP server to an HTTP client, the former will be referred to as the ‘sender’, and the latter as the ‘receiver’.

TCP is a sliding window protocol. It therefore limits the amount of outstanding data to a size called the sender window. The sender window is set to the minimum of two parameters:

1. The receiver advertised window, which reflects the amount of free space the receiver has in its receive-buffer.

2. The sender congestion window *(cwnd)*, as determined by the TCP congestion control scheme.

TCP congestion control has three modes: slow-start, congestion-avoidance and fast-retransmit. The main difference between the modes is the increasing rate of *cwnd*. The sender enters slow-start when it starts to send data or after a timeout. It initializes *cwnd* to a size of one segment, and increases it by size of one segment whenever a new ACK is received. This results in doubling *cwnd* every round-trip time *(RTT)*. The shift from slow-start to congestion-avoidance is made when *cwnd* reaches the congestion window threshold *(ssthresh)*. This threshold is supposed to indicate that the sender window has nearly reached the network capacity, and that from this stage *cwnd* should be increased more carefully. Hence, during congestion-avoidance *cwnd* is only linearly increased: by $\frac{1}{cwnd}$ for every received ACK, which is equivalent to one segment for every RTT. At connection setup, the value of *ssthresh* is initialized to the maximum window size, 65536 bytes. It is updated to one half of the sender window in two cases: following a timeout, or when the sender enters fast-retransmit.

The third mode of TCP congestion-control is *fast-retransmit*. The sender enters fast-retransmit mode upon receiving 3 duplicate ACKs. After the third duplicate ACK is received, the sender retransmits the missing segment, sets *ssthresh* to one half of the sender window and

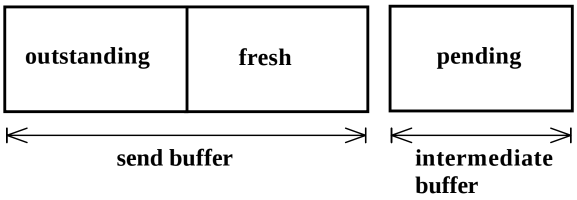

Figure 2: Stages in the lifetime of a data segment.

reduces *cwnd* to $\lfloor \frac{cwnd}{2} \rfloor + 3$. The sender exits fast-retransmit when an ACK for the retransmitted segment is received. Upon exiting fast-retransmit, the sender enters congestion-avoidance and *cwnd* is set to *ssthresh*, i.e. to one half of the window size before fast-retransmit was entered.

We distinguish between 3 different stages in the lifetime of a data segment, from the moment it is read from the server disk until it is acknowledged by the client and discarded from the send-buffer (see Figure 2).

Definition 1:

1. A *pending segment* is a data segment that was fetched from the disk into the main memory but has not yet been copied into the send-buffer.
2. A *fresh segment* is a data segment that was copied into the server send-buffer but not yet sent.
3. An *outstanding segment* is a data segment that was sent but not yet acknowledged.

3.2 The Optimal Send-Buffer Size

The optimal size of a send-buffer for a TCP connection is the smallest size that still enables the connection to use the maximum available bandwidth, i.e. to send a new segment whenever TCP flow control allows. The difficulty in finding the optimal size is that it changes during the connection lifetime, even if the network conditions do not change. The optimal size is directly affected by the following three factors:

1. The sender *cwnd*.

2. The connection RTT.

3. The server *ReadTime*. We define *ReadTime* as the time needed to access, read and transfer a new block of data from a file into the send-buffer. In most cases files are stored on a secondary storage media, in which case *ReadTime* is mainly affected by the media access time and the server load.

Since new data can be fetched from the disk only once per *ReadTime*, the send-buffer should contain enough fresh segments to enable all possible transmissions of new segments to take place during a *ReadTime* period. The maximal number of fresh data segments that can be sent during a *ReadTime* period depends on the number of data segments that are outstanding when the file access is invoked, and on the congestion control mode. For example, suppose that $ReadTime < RTT$ and assume that at time t a new file access is invoked. If at that time there are n outstanding data segments and the connection is in slow-start, then until time $t+ReadTime$ no more than n ACKs can be received. Since during slow-start a receipt of a new ACK triggers the sending of two new data segments, no more than $2 \cdot n$ fresh data segments can be sent before the file access is completed. Therefore, in order to avoid unnecessary delays, at time t the send-buffer should contain not only the n outstanding segments, but also additional $2 \cdot n$ fresh segments. From the same considerations, the maximum number of fresh segments needed during time interval $[t + RTT, t + 2 \cdot RTT)$ is $4 \cdot n$ segments. Hence, the file access triggered at time t should fetch at least $4 \cdot n$ segments.

The proposed mechanism for local optimization is based on the above observation. Its main idea is to manage a dynamic send-buffer size, that adapts to the actual condition of the connection and the network.

3.3 A Scheme For Local Optimization

In what follows, a scheme for local optimization of buffer allocation is presented. Denote by $cwnd[t]$ and $ssthresh[t]$ the values in segment units of the sender's variables *cwnd* and *ssthresh*[1] at time t. Throughout the paper, we usually refer to the values of variables *after* some event

---

[1] Although *cwnd*, *ssthresh*, and the send-buffer water marks are measured in bytes, it is more convenient to consider them in segment units.

takes place. For instance, if an ACK is received at time t, then $cwnd[t]$ indicates the value of $cwnd$ after the ACK is processed. In those rare cases where we refer to the value of a variable *before* the event, we shall use $[t^-]$ rather than $[t]$.

For the formal discussion we define $next(cwnd)[t]$ as follows:

$$next(cwnd)[t] = \begin{cases} \min(2 \cdot cwnd[t], ssthresh[t]) & \text{if } cwnd[t] < ssthresh[t] \text{ (slow-start)} \\ \lfloor cwnd[t] \rfloor + 1 & \text{otherwise (congestion-avoidance or fast-retransmit)} \end{cases}$$

We also denote by $next^2(cwnd)[t]$ the value of $next(next(cwnd)[t])[t]$. For simplicity, we shall use $next[t]$ as a shortened form for $next(cwnd)[t]$, and $next^2[t]$ as a shortened form for $next^2(cwnd)[t]$. Intuitively, if the number of outstanding segments at time t is $cwnd[t]$, then $next[t]$ is an upper bound on the number of fresh segments that can be transmitted during time interval $[t, t+RTT)$, and $next^2[t]$ is an upper bound on the number of fresh segments that can be transmitted during time interval $[t + RTT, t + 2 \cdot RTT)$.

Following is the proposed local optimization scheme for the case where $readTime < RTT$. A generalization for this scheme is described later.

Procedure 3.1 Local Optimization Scheme (*LOS*)

1. If at time t $cwnd$ changes[2], set `sb_hiwat`= $\lfloor cwnd[t] \rfloor + next[t]$.
2. The first file access should fetch from the disk at least 3 segments.
3. Any other file access, invoked at time say t, should fetch from the disk at least $next^2[t]$ segments.

Consider as an example a connection in congestion-avoidance, as depicted in Figure 3. Assume that at time t_0 $cwnd= 2.5$, and there are 3 fresh segments and a pending one. At time t_1 an ACK is received for segment 0. Hence, $cwnd$ is incremented by $\frac{1}{cwnd} = 0.4$ to 2.9, the acknowledged segment is deleted from the send-buffer and the pending segment is appended. Since there are no more pending segments, then according to the server model described in Section 2, a new file access is invoked. Following rule 3 of Procedure 3.1, the file access is of size $next^2[t_1] = next(next(2.9)) = 4$ segments. At t_2, an ACK for segment 1 is received and $cwnd$

---

[2] Actually, the frequency of `sb_hiwat` updates can be much lower, as explained later.

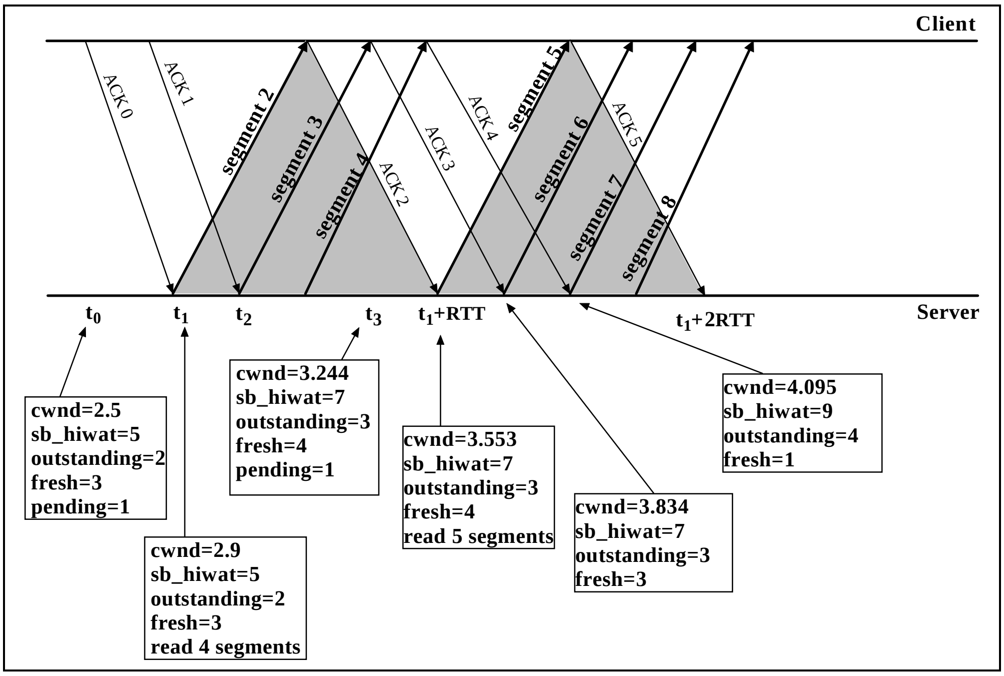

Figure 3: An example for the Local Optimization Scheme.

increases by $\frac{1}{2.9}$ to 3.244. Hence, 2 fresh segments can be transmitted (segments 3 and 4 in the figure), increasing the number of outstanding segments to 3 and leaving only one fresh segment in the buffer. The file access is completed at $t_3 = t_1 + ReadTime$. Following rule 1 of Procedure 3.1, $\texttt{sb\_hiwat}[t_3] = \lfloor cwnd[t_3] \rfloor + next[t_3] = \lfloor 3.245 \rfloor + next(3.245) = 3 + 4 = 7$. Since just before t_3 there are 3 outstanding segments and only 1 fresh segment, $7 - 3 - 1 = 3$ segments from the read chunk can be appended to the send-buffer, and 1 segment remains pending. At time $t_1 + RTT$ an ACK for segment 2 is received, and therefore $cwnd$ becomes $3.245 + \frac{1}{3.245} = 3.553$. The acknowledged segment is deleted from the send-buffer and therefore the pending segment is inserted. Hence, a new file access, of $next^2(3.553) = 5$ segments, is invoked. Note that the number of fresh segments sent during time period $[t_1 + RTT, t_1 + 2 \cdot RTT)$ is equal to the number of segments fetched in the file access invoked one RTT earlier, at t_1.

In the following we present simulation results for the new scheme. The simulations were performed with version 1 of the network simulator *ns* [10]. We tested connections over 1-Mbps link with an RTT of 150 msec and a configurable uniform loss. The increase in the loss rate reflects a decrease in the network available bandwidth. Each connection transferred 500 Kbytes (1000 segments of 512 bytes) of data. We compared the performance of a connection that uses the proposed local optimization scheme *(LOS)* to the performance of a connection that uses fixed sized `sb_hiwat` of 8, 16 or 32 Kbytes. During the simulations, a connection that uses LOS receives as much buffers as demanded by the scheme. We have averaged the results of 50 simulation runs, each with a different loss-generation seed. In order to achieve the burstiness effect of segment losses, we have increased the loss probability of a segment that follows a lost segment.

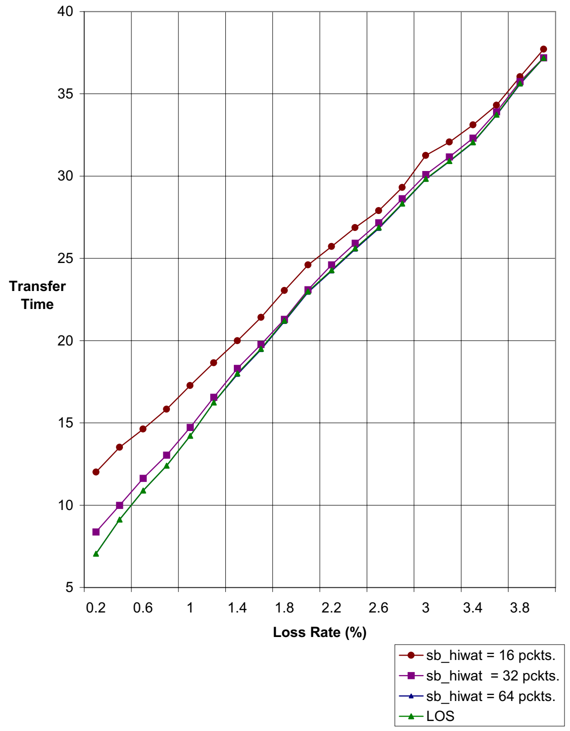

Figure 4: Average time for 500 Kbyte transfer.

Figure 4 depicts the average time needed to complete a 500 Kbyte transfer, as a function of the connection loss rate. LOS achieves the best utilization of the available bandwidth. The same utilization is achieved when a 64-segment (32 Kbytes) buffer is used. For a 16-segment and 32-segment buffers, the throughput is smaller. However, the real advantage of LOS is evident from Figure 5 that shows the average amount of buffers used during the transfer. While LOS detects changes in the available bandwidth and allocates less buffer space accordingly, the scheme that uses a fixed `sb_hiwat` always allocates the maximum buffer space.

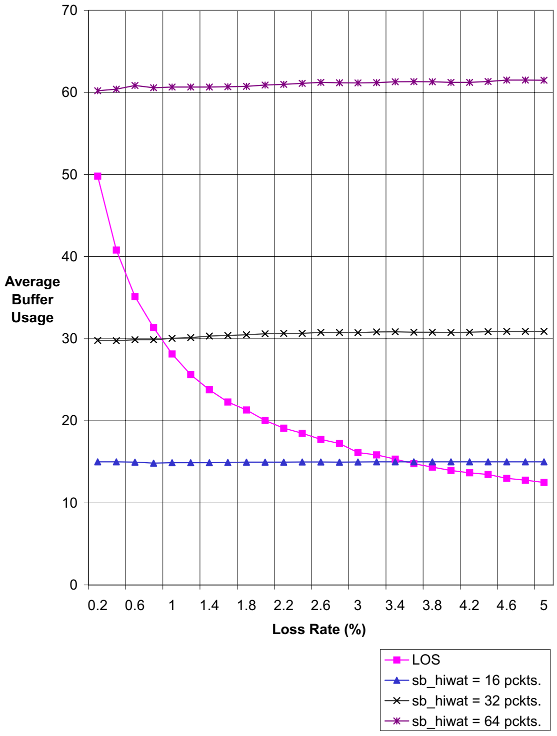

Figure 5: Average buffer usage.

LOS intends to avoid cases where TCP output procedure wants to send a segment, but such a segment does not exist since it was not yet fetched from the disk. In what follows, such an event is referred to as *buffer miss*. Figure 6 depicts the average number of buffer misses encountered during the simulations. The schemes that allocate a 16-segment buffer and a 32-segment buffer had many buffer misses under a low loss rate. The scheme that allocates a 64-segment buffer experienced, on the average, less than one buffer miss, whereas with LOS no buffer miss is encountered.

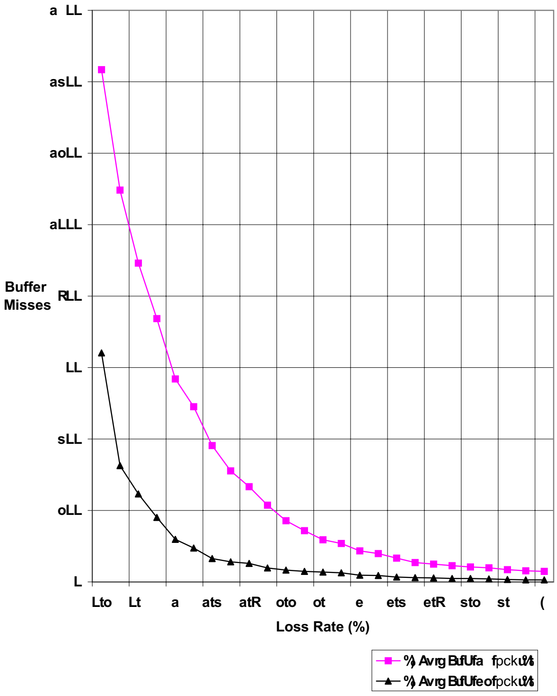

Figure 6: Average number of buffer misses.

Disk access time is composed of three basic delay components. The first component is the time spent while waiting on the disk controller's queue. The second component is the sum of the seek time and the rotation latency. An average value for this component is about 15 milliseconds. The last component is the transfer time, namely the time needed to physically read the data from the disk. Therefore, in environments such as high bandwidth LANs or broadband access networks (with servers located in a local POP), disk access time can be considerably greater than RTT.

In the following we present a generalization of LOS that handles the case where $ReadTime \geq RTT$. Define T_{fact} as $\lceil \frac{ReadTime}{RTT} \rceil$. Previously we saw that when $T_{\text{fact}} = 1$, it is sufficient to hold in the send-buffer additional $next[t]$ fresh segments in order to guarantee maximum throughput, i.e. no buffer miss. When $T_{\text{fact}} = k$, the send-buffer should contain $next[t] + next^2[t] + \ldots + next^k[t]$ additional segments in order to allow the sender to work at maximum possible speed during a period of $T_{\text{fact}} \cdot RTT$. As an example, consider Figure 7 that describes a connection in congestion-avoidance. Assume that at time t' $T_{\text{fact}} = 2$ holds, and that $cwnd = N$. Therefore, at this time `sb_hiwat`$=$ $cwnd + next[t'] + next^2[t'] = N + (N+1) + (N+2)$. Suppose that at t' there is only one pending segment. Therefore, at time t_0, when a new ACK that acknowledges outstanding data is received, the pending segment is inserted into the buffer and a new file access is invoked. Until time $t_0 + T_{\text{fact}} \cdot RTT$, the time when the file access has completed, no more than $(N+1) + (N+2)$ new segments can be transmitted. Hence, buffer miss is not possible. Note that the file access initiated at t_0 should read at least $next^3[t_0] + next^4[t_0] = (N+3) + (N+4)$ new segments.

The proposed solution requires an estimation of both $ReadTime$ and RTT. RTT estimation is discussed in Appendix A. In contrast to RTT, disk access time is highly dependent on the specific disk system. Most advanced servers today employ RAID-based systems [5]. A summary of the research performed in the area of RAID performance evaluation, in terms of utilization, response time etc., is given in [11].

3.4 Implementation Issues

The local optimization scheme requires minor adjustments to the server model presented in Section 2. This section summarizes these adjustments. Procedure 3.2 is a modified version of procedure 2.1.

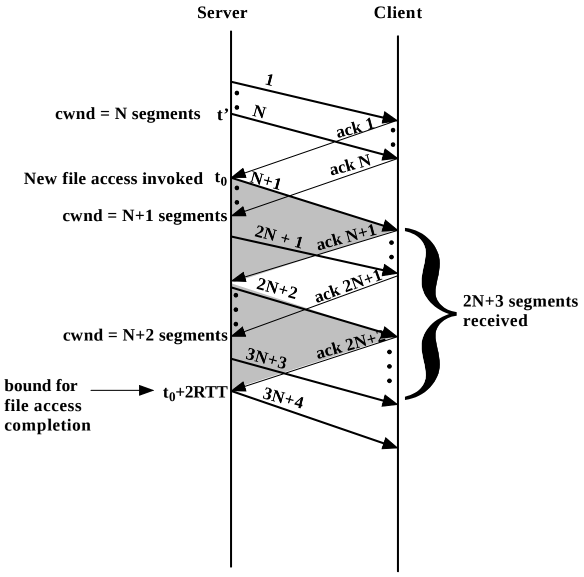

Figure 7: Time factor consideration

Procedure 3.2

```
size-of-read = initialValue;
array = allocateArray(size-of-read);
while (nbytes=fread(requested-file, array, size-of-read, ...)) {
    ...
    send(socket,..., array, nbytes);
    free(array);
    size-of-read = calcNextRead(socket);
    array = allocateArray(size-of-read);
    ...
}
```

The main difference is the call to a new function, calcNextRead, that computes $next^2[t]$ for the connection. The modified server routine calls calcNextRead in order to determine the size of the next file access. A new intermediate buffer with the proper size is allocated for every file access.

Procedure 3.3 is a modified version of procedure 2.2, describing the operation of the socket layer function `sosend`.

Procedure 3.3

```
repeat {
    updateBufferSize();
    while (sb_hiwat-sb_cc<sb_lowat) {
        wait;
        updateBufferSize(); }
    fill network buffers;
    call TCP output procedure;
} until all data in the block has been transferred to the send-buffer;
```

The new procedure *updateBufferSize* is called in order to compute $next[t]$ and set `sb_hiwat` to $\lfloor cwnd \rfloor + next[t]$, according to the optimization scheme. `Sb_hiwat` is recomputed in two cases: upon entering a new iteration of the loop, and when a process is resumed after waiting for buffer space. Note that unlike 2.2, the modified procedure can add multiple buffer units to the send-buffer during a single iteration.

4 Global Optimization

Suppose there is a limit N on the number of buffers the kernel assigns for TCP connections. In such a case, the kernel might not be able to allocate to each connection the amount of buffer space specified by LOS. This raises the need for a global optimization scheme, that assigns the N available buffers to the active connections according to some policy. The naive allocation policy, employed by HTTP servers today, is to give each active connection an equal share of the buffer space. Namely, every established connection has the same `sb_hiwat`. This naive approach yields low aggregated throughput because it does not take into account the variance in the bandwidth-delay products and RTTs of different connections. For example, a connection over a 6-Mbps ADSL link is allocated the same buffer size as a connection over a 14.4-Kbps analogue modem. The connection that uses an ADSL link is most likely to under-utilize its available bandwidth due to buffer misses, whereas the connection that uses a low-speed modem is allocated more buffer space than it actually needs in order to fully utilize its available bandwidth.

Following is the description of two proposed allocation policies. Both policies employ LOS in order to compute the optimal buffer space every connection needs. A queue for connections that are waiting for more buffer space is defined. As long as there is no shortage in buffers, the queue remains empty. If a connection needs a buffer when the buffer pool is empty, the associated process is suspended and enqueued. The main difference between the two policies is their queuing discipline:

1. Round Robin *(RR)*: the queue is managed according to first in first out (FIFO) discipline. When a buffer is released and the queue is not empty, the first connection is dequeued. The connection appends this buffer to the send-buffer. If it needs more buffers, it is enqueued once again. Since LOS guarantees that a connection never requires more buffers than what it actually needs, this policy is equivalent to the implementation of max-min fairness.

2. Priority Queuing *(PQ)*: the queue is managed as a priority queue. The priority of each queued connection is set to $min(p, \frac{i}{k})$, where k is the number of buffers the connection already holds, i is the connection `sb_hiwat` as computed by LOS, and p is the maximal priority for connections holding buffers. A connection holding no buffers is granted with priority $p + 1$. This is because such a connection is most likely to be able to use the new buffer for the transmission of a new segment, thus contributing to the aggregated throughput. This scheme ensures fairness in the sense that all connections get the same percentage of the buffer space they need.

We used *ns* to simulate these allocation policies and evaluate their performance. The simulated network consists of one server node connected to 100 different client nodes through disjoint links. Two types of links are defined: 1 Mbps links and 0.1 Mbps links. The round trip associated with every link is 150 msec, and *readTime* is 50 msec. The segment loss rate is 0.01, and effect of burstiness is achieved by increasing the loss probability of a segment that follows a lost segment. The connections established over the 1 Mbps links are referred to as *fast* connections whereas those established over 0.1 Mbps are referred to as *slow* connections. The server task on every simulation run is to transfer 250 Kbytes (500 segments of 512 bytes) over 50 different connections, assuming it is capable of handling up to 10 connections simultaneously[3]. The following results reflect the average of 50 simulation runs, each with a different pseudo-random generator seed. When the naive policy is tested with a server buffer pool of N buffers, `sb_hiwat` of all connections is set to $\frac{N}{10}$.

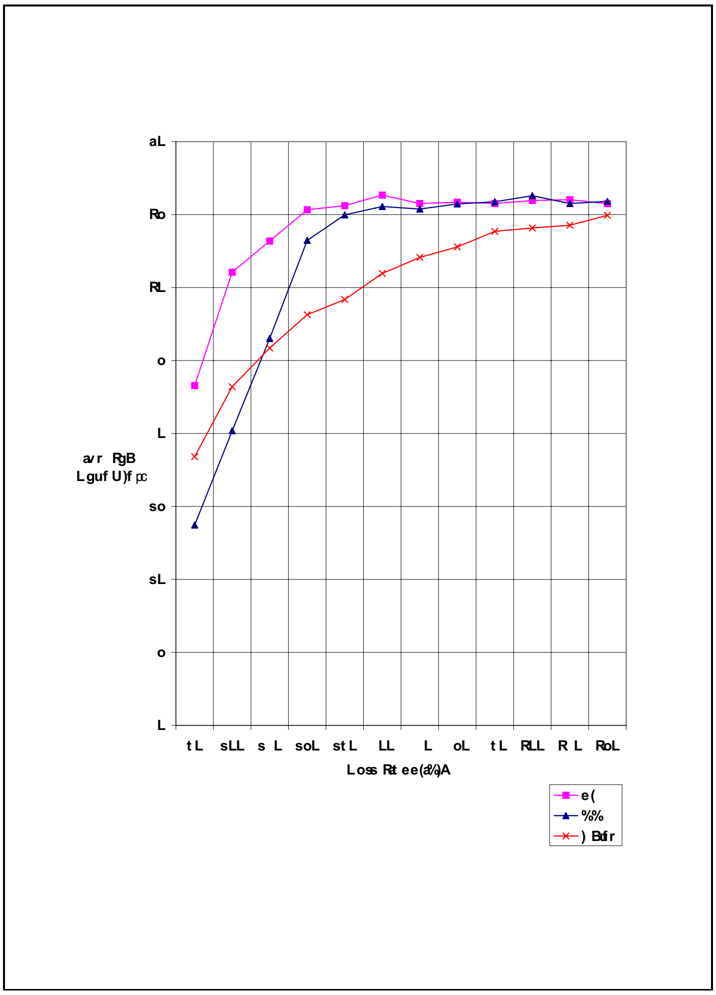

Figure 8: Average bandwidth of fast connections.

---

[3] A relatively small number of connections was used in order to bound simulations run time. The tested server buffer pool sizes were selected respectively. However, a larger number of connections and larger buffer pools are to derive similar results.

Figure 8 depicts the average bandwidth of the fast connections as a function of the server buffer pool size. It is evident that PQ achieves considerably better performance for every buffer pool size, and that for a large pool size PQ and RR achieve the same performance. When the pool size is very small, PQ is superior whereas the performance of RR is worst than the performance of the naive policy. The reason for this is as follows. Under the naive policy, buffers released during connection termination are returned to the pool and are not claimed by any other connection. Hence, a new connection can be immediately allocated up to `sb_hiwat` of buffers. On the other hand, when RR is used and the pool size is small, released buffers are always allocated to existing connections, and therefore a new connection is likely to wait longer before getting buffers and contributing to the aggregated bandwidth. PQ avoids this problem because a new connection is most likely to have a higher priority than the "old" connections, since it either holds no buffers at all or just a small fraction of what it needs. With slow connections, all policies achieved the maximum available bandwidth for all reasonable buffer pool sizes. This is predictable since these connections require relatively small buffer space in order to fully utilize the available bandwidth.

Figure 9 shows the average buffer space used by the server during the simulation, i.e. the sum of buffers used by all active connections (fast and slow), as a function of the buffer pool size. While RR and PQ never use more buffers than needed, the naive policy frequently allocates more buffers than needed. For example, when the buffer pool size is 350 segments, the naive policy allocates the connections 100% more buffers than PQ does but nevertheless, achieves 4% less bandwidth for the fast connections. Hence, a server employing the naive policy needs a considerably bigger memory space in order to achieve maximum performance. A server running PQ or RR can use the extra buffer space to increase the number of accommodated connections, or for other important tasks, like file caching. Furthermore, such methods are suitable for embedded systems where high utilization of the available memory is essential.

The fairness of the proposed allocation schemes was tested using the fairness index proposed in [9]. This index is computed as $\frac{[\sum x_i]^2}{n \cdot \sum x_i^2}$, yielding a value of 1 for a perfectly fair policy and a smaller value for less fair policies. Using this index for the measured throughputs, $\{x_1, x_2, \ldots, x_n\}$, of connections from the same type, we found all the proposed schemes to be more than 90% fair.

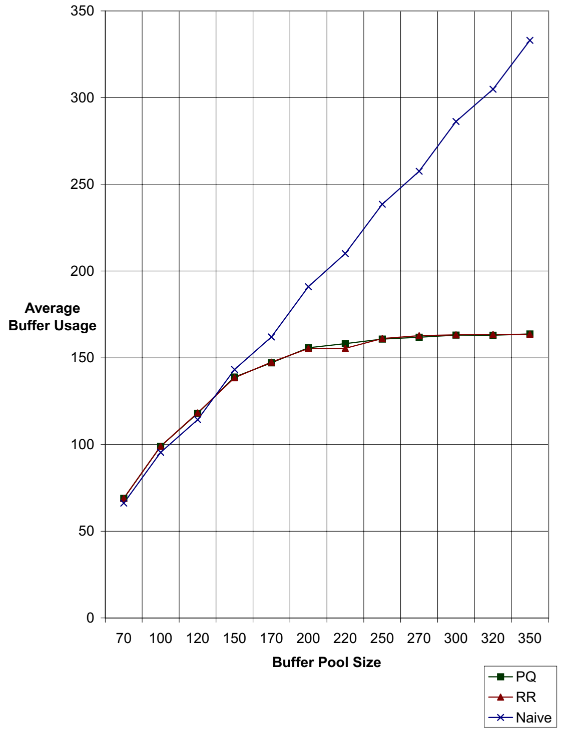

Figure 9: Average buffer usage

5 Related Work

in [7], another proposal for efficient buffer allocation by a TCP server, denoted the *PSC Solution*, is presented[4]. This solution can also be classified as a local optimization method that dynamically tunes TCP send-buffer size according to the connection condition, and a global

---

[4] The two researches were conducted independently of each other, and the two proposals were introduced almost in parallel in October 1998.

optimization method that ensures that the global amount of buffers is shared fairly among all active connections. Although the general approach is similar, there are a few major differences between the two solutions as described in the following section.

The main difference in the local optimization method is that the PSC solution ignores *ReadTime*, which is the time required to fetch a new block of data from the secondary storage. As mentioned earlier, ignoring *ReadTime* might lead to throughput degradation in a broadband LAN or a broadband access network, where the RTT is often smaller than *ReadTime*. Furthermore, the method presented in this paper maximizes the disk goodput by optimizing the size and number of disk accesses in the sense that only the data blocks with maximal contribution to the overall throughput are fetched from the disk.

The global optimization method presented in [7] and in a subsequent work [12], is based on a fairness routine that periodically scans all connections and, when required, updates their send-buffer size limit such that max-min fairness is achieved. A hidden assumption is that the sum of all the buffer sizes of all the connections is smaller than or equal to the size of the global buffer pool. Under such an assumption, it cannot happen that a connection tries to allocate a buffer when the pool is empty. However, in a server environment there are other clients, except TCP connections, that need to access the pool of network buffers, such as UDP-based applications and the ICMP mechanism. Therefore, in order to guarantee that there are always free buffers, the PSC solution requires bigger buffer pools. In contrast, the solution proposed in this paper does not encounter this problem since it queues unfulfilled buffer requests. When a buffer is released to the pool, it is immediately allocated to the connection at the head of the queue. Although it requires a more complicated logic than in the PSC solution, this gives our approach several additional advantages as follows. First, as opposed to the PSC solution, our global optimization method does not require to periodically scan all the connections in order to enforce fairness. This periodic scan can exploit considerable amount of processing power when the connection load is in an order of magnitude bigger than the load tested in [7]. Second, and more important, our method is generic enough to support several global policies in addition to max-min fairness. This is an important advantage since we have shown that other fair policies, such as PQ, may achieve better performance than max-min fairness.

6 Conclusion

We have described new dynamic methods for efficient TCP buffer allocation in an HTTP server. In the first part of the paper, we proposed a scheme for local optimization. The basic idea of the proposed scheme is to dynamically adjust the TCP send-buffer size to the connection and server characteristics. The main advantage of this scheme is that it enables to accurately estimate the minimum amount of buffers a connection needs in order to achieve maximum throughput. Such a scheme is of increasing importance due to the large scale of connection characteristics in the Internet.

The second part of the paper discussed the case where the amount of buffer space needed by TCP connections is larger than the size of the server buffer pool. In this context, two allocation schemes for global optimization were proposed: RR and PQ. When compared to the allocation policy employed by a typical HTTP server, the schemes were shown to achieve better utilization of the server memory and much higher aggregated server throughput. The optimization methods require only minor changes and only to the server code.

A RTT Measurements

TCP measures the round trip time *(RTT)* of every connection in order to compute the retransmission timeout *(RTO)*. As previously shown, LOS uses the estimated RTT in order to evaluate T_{fact}. However, in typical TCP implementations, the measurement of RTTs is inaccurate for two reasons:

- A typical TCP implementation measures the round-trip time of only one segment at a time. The difference between the sampling rate and the segment transmission rate, has been shown to result in unreliable RTT measurements (e.g. [8]).
- For the RTT measurements, a typical TCP implementation uses a coarse-grained clock, with granularity of 0.5 seconds. This imposes a serious limit on the measurement precision.

Since inaccurate measurement of RTT has a significant negative effect on the connection performance, i.e causing either unnecessary or late retransmissions, the following improvements were proposed. In [8], where TCP extensions for high speed networks are discussed, it is proposed to use the time-stamp option in order to increase the measurement precision. The idea is that every segment will carry a time-stamp in the TCP options field. The receiver copies the time stamp into the acknowledgement and, therefore enables the sender to compute the round-trip time accurately for every segment. In [3], it is proposed to keep the transmission time of every outstanding segment, thus enabling to compute the RTT of almost every segment upon its acknowledgement. Unlike the previous method, this one requires no cooperation from the receiver side. Both solutions are shown to yield much more accurate RTT measurements.

Even if RTT measurement rate increases, the granularity of the TCP clock still imposes a limitation, since T_{fact} can not be computed for connections that experience RTTs considerably smaller than 0.5 seconds (in such cases the measured RTT of the connection is often 0 clock ticks). The solution is either to use a clock with a better granularity or to enhance the precision of the smoothed *RTT (srtt)* as proposed in [2]. A much simpler, though less accurate, solution is to use a pre-defined T_{fact}, e.g. 1 or 2, for connections whose RTT is smaller than 0.5 seconds. Note that in general, enhancing clock granularity has been shown to improve TCP performance (e.g. [6]).

Acknowledgments

This research was supported by the Technion, Israel Institute of Technology, and was conducted, as part of a Master thesis, in the Dept. of Computer Science.

References

[1] Apache HTTP server project. http://www.apache.org.

[2] L. S. Brakmo and L. L. Peterson. Performance problems in bsd4.4 tcp. *Computer Communication Review*, 25(5):69–86, Oct. 1995.

[3] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, October 1995.

[4] A. Cohen and R. Cohen. A dynamic approach for efficient TCP buffer allocation. In *Proceedings of the Seventh International Conference on Computer Communications and Networks (ICCCN)*, pages 817–824, October 1998. Also, a Technical Report LCCN9805, Dept. of Computer Science, Technion August 1998.

[5] Garth Gibson David A.Patterson and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM conference on Management of Data (SIGMOD).*

[6] S. Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5):10–23, October 1994.

[7] J. Mahdavi J. Semke and M. Mathis. Automatic TCP buffer tuning. *Computer Communication Review*, 28(4), October 1998.

[8] V. Jacobson, R. T. Braden, and D. A. Borman. TCP extensions for high performance. Technical report, RFC 1323, May 1992.

[9] R. Jain, D. Chiu, and W. Hawe. A quantative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, DEC Research Report, Sept. 1984.

[10] S. McCanne and S. Floyd. *NS Network Simulator.* http://www-nrg.ee.lbl.gov/ns, 1995.

[11] Garth A. Gibson Peter M. Chen, Edward K. Lee and Randy H. Katz. RAID: high-performance, reliable secondary storage. Technical Report UCB/CSD-93-778, University of California at Berkeley, 1993.

[12] J. Semke. Implementation issues of the autotuning fair share algorithm. Technical Report CMU-PSC-TR-2000-0002, Pittsburgh Supercomputing Center, May 2000.

[13] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley Publishing Company, 1995.