TCP Implementation Enhancements for Improving Webserver Performance

Mohit Aron Peter Druschel
{aron,druschel}@cs.rice.edu

TR99-335
Department of Computer Science
Rice University
Houston, TX 77005

Abstract—

This paper studies the performance of BSD-based TCP implementations in Web servers. We find that lack of scalability with respect to high TCP connection rates reduces the throughput of Web servers by up to 25% and imposes a memory overhead of up to 32 MB on the kernel. We also find that insufficient accuracy in TCP's timers results in overly conservative delays for retransmission timeouts, causing poor response time, low network utilization and throughput loss. The paper proposes enhancements to the TCP implementation that eliminate these problems, without requiring changes to the protocol or the API. We also find that conventional benchmark environments do not fully expose certain significant performance aspects of TCP implementations and propose techniques that allow these benchmarks to more accurately predict the performance of real servers.

Keywords—**Internet, TCP, Webserver, Timers**

I. Introduction

With the widespread growth in the use of the World Wide Web (WWW), webservers are expected to keep up with the ever increasing performance demands of the user community. The WWW protocol, HTTP[1, 2], uses TCP[3] (Transmission Control Protocol) as its underlying transport protocol for data delivery. The TCP implementation distributed with 4.4BSD[4] is incorporated in several operating systems (FreeBSD, NetBSD, Digital UNIX, IBM's AIX etc) and is one of the fastest and most efficient. Moreover, the BSD implementation has served as a model for several other operating systems, including ones that are not UNIX-based. This paper focuses on enhancements in BSD-based TCP implementations aimed at improving performance in busy webservers.

A busy webserver manages thousands of TCP connections. This paper shows that the BSD TCP implementation does not scale with high rates of TCP connection establishments and teardowns. Measurements with real workloads indicate that this lack of scalability degrades webserver throughput by up to 25% and imposes a memory overhead of up to 32 MB on the kernel of the webserver machine. We present results that indicate that these overheads are caused by the management of connections in TCP's TIME_WAIT[5] state. We propose implementation enhancements that are capable of reducing these overheads. We also find that conventional benchmark environments do not fully expose these overheads and we propose techniques that enable benchmarks to more accurately predict the performance of real servers.

This paper also discusses limitations due to the use of a relatively coarse-grained clock by TCP, both for time measurement as well as for scheduling timer events. These include overly conservative retransmission timeouts (RTO) and inability to measure high bandwidths accurately using the packet-pair[1] technique[6, 7]. Overly conservative retransmission timeouts and the resulting degradation in server response time has also been observed in past work by Brakmo and Peterson[8]. We address this by proposing a timer framework that decouples the time measurement clock from the event scheduling clock. The principal advantage afforded by this decoupling is the ability to use a fine-grained clock (micro to nano second granularity) for interval measurements. Our results indicate that this achieves up to a factor of 15 tighter delays in TCP retransmission timeouts. In addition, the finer granularity afforded by the time measurement clock can be used by TCP to measure bandwidth accurately on high bandwidth networks using techniques like the packet-pair algorithm.

Although our study is made in the context of busy webservers, we expect that our techniques also benefit other applications like ftp and telnet that use TCP.

The rest of this paper is organized as follows. Section II provides some background on the implementation of timers in TCP and the TIME_WAIT state. Section III describes the experimental environment used for obtaining the results reported in this paper. In Section IV we quan-

[1] The packet-pair technique estimates network bandwidth by observing the spacing between closely spaced packets.

tify the CPU and memory overheads in a busy webserver and discuss the limitations due to insufficient accuracy in TCP timers. In Section V we present techniques to reduce the CPU and memory overhead associated with TCP TIME_WAIT processing. Section VI proposes our timer framework that decouples the time measurement clock from the event measurement clock. In Section VII we discuss related work and we conclude in Section VIII.

II. Background

This section provides the background information on the implementation of TCP timers in BSD-based network stacks and on the TCP TIME_WAIT state.

A. Implementation of Timers in BSD TCP

Connection-oriented protocols like TCP have to rely on several per-connection timers for correct operation. To maintain portability across a wide range of hardware platforms, the BSD implementation of TCP expects minimal timer support from the operating system. It expects the operating system to provide it with two periodic "software interrupts" – one of these interrupts invokes a TCP handler (called fasttimeo) every 200ms while the other invokes another handler (called slowtimeo) every 500ms. These two interrupts constitute all of the timing services that TCP expects of the operating system; all TCP timer mechanisms (round-trip time estimation and various TCP timeouts) are implemented by the TCP module using these interrupts.

Seven timers are maintained by TCP for each connection: 1) *connection-establishment* timer, 2) *retransmission* timer, 3) *delayed* ACK (acknowledgment) timer, 4) *persist* timer, 5) *keepalive* timer, 6) *FIN_WAIT_2* timer, 7) *TIME_WAIT* timer. The delayed ACK timer is implemented by the fasttimeo function, while all the other timers are implemented by the slowtimeo function.

Except for the delayed ACK timer, all other timers are set by setting corresponding integer variables associated with the connection. Similarly, canceling a timer simply requires setting the value to zero. The slowtimeo function linearly scans through the list of all connections and decrements these integer variables, executing the associated handler if the value reaches zero. Thus setting and canceling timers is optimized at the expense of checking and firing a timeout. The fasttimeo function also makes a linear scan over all the connections, sending an ACK for every connection that has one pending.

B. TCP TIME_WAIT state

TCP requires that the end that closes a connection first (active close) undergo a TIME_WAIT state; it serves two main purposes – 1) to detect and discard old packets from the connection, and 2) to resend the acknowledgment for the connection termination packet (FIN) from the other end in case the former got lost. In the case of HTTP, the server does the active close which results in the server TCP to go through the TIME_WAIT state. The state associated with a connection in the TIME_WAIT state is discarded after the passage of time corresponding to twice the MSL (maximum segment lifetime), typically two minutes in the current Internet. However, most operating systems (including BSD) set the MSL to 30 seconds, requiring a TIME_WAIT period of 1 minute.

III. Experimental Environment

This section describes the experimental environment used for obtaining the results presented in the rest of this paper. We emulate both LAN (local area network) as well as WAN (wide area network) benchmark environments.

Our setup consists of four 300MHz Pentium-II machines running FreeBSD-2.2.6 operating system, each configured with 128 MB of RAM and connected to each other through a switched 100Mbps Ethernet. This configuration is depicted in Figure 1. We ran the Apache-1.3.3[9] webserver on one of the machines while the other three machines ran a client program that generated HTTP requests for the Apache webserver. Our client program was based on the S-client architecture[10] which is an event-driven program that simulates several HTTP clients. The program generates HTTP requests as fast as the webserver can handle them.

A. Delay Router

Some of our experiments modeled a WAN environment. Emulating a WAN environment provides results that are more representative of servers on the Internet. For this purpose, we used a *delay router* to emulate WAN propagation delays (see Figure 2) in a LAN test-bed. One of the main challenges in the design of the delay router is to preserve the temporal spacing of packets on the network.

The design of our delay router is similar to that described by Banga and Druschel[10]. The delay router was implemented by modifying the forwarding mechanism in the TCP/IP protocol stack. Vanilla IP forwards a packet to another machine as soon as it is received on an interface. In contrast, the delay router queues packets in a kernel data structure to be forwarded after a specified delay. A dedicated kernel process continuously checks for packets that are to be forwarded.

Continuous polling is used instead of a periodic interrupt driven check in order to conserve the temporal spacing of packets on the network. In contrast, a periodic interrupt based scheme introduces burstiness in the out-

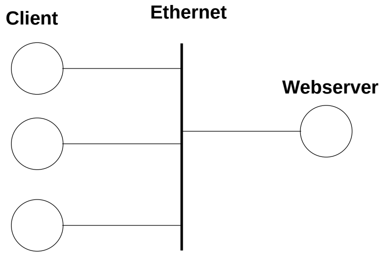

Fig. 1. LAN configuration

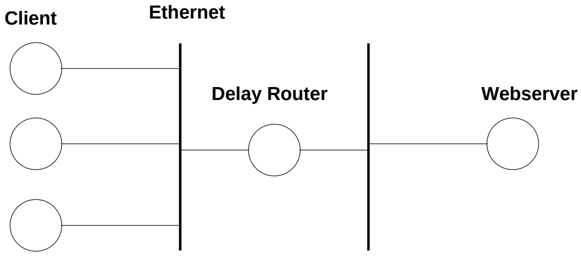

Fig. 2. WAN configuration

going packet train, because all packets due for transmission within a check period are transmitted simultaneously. Even with a clock period of 10ms, which corresponds to the interrupt-clock frequency of many commonly-used operating systems, up to 84 back-to-back 1500 byte packets may be sent during each check period if the bandwidth is 100Mbps. This issue is important because TCP performance is known to be affected by phenomena such as ACK compression[11], which also disturbs the timing of packets in the network.

IV. TCP Implementation Problems

This section evaluates the implementation problems in BSD TCP. We show that the implementation of timers in TCP can cause significant CPU and memory overhead in a busy webserver. We discuss why conventional benchmark environments do not expose this overhead and propose a technique that allows benchmarks to more accurately predict the performance of real servers. In Section V we present enhancements to the timer implementation that reduce the aforementioned overhead.

We also discuss limitations due to the use of a relatively coarse-grained clock by TCP, both for time measurement as well as for event scheduling. These include overly conservative retransmission timeouts and inability to measure high bandwidths accurately. In Section VI, we propose a timer framework that addresses these limitations.

A. Measuring the overhead

This subsection presents experimental benchmark results that measure the CPU and memory overheads of a busy webserver. We argue that conventional benchmark environments with limited number of client machines cannot fully expose the overheads that are representative of a real webserver.

Figure 3 shows the throughput of the Apache webserver when the client machines in Figure 1 repeatedly request a cached 6 KB file from the webserver. The file size chosen is representative of average web content on the Internet[12, 13]. Our results show that the Apache webserver attains a throughput of about 800 conn/s with conventional client benchmark.

We instrumented the kernel on the webserver machine to also measure the maximum time taken in the slowtimeo and fasttimeo functions mentioned in Section II-A. These only include the time for scanning the list of connections – they do not include the time for actually executing timeouts. The maximum CPU overhead imposed by the slowtimeo and fasttimeo functions was calculated from these and is also depicted in Figure 3. Also shown are the maximum number of connections in TIME_WAIT and the peak kernel dynamic memory allocation during the experiment.

Since the TIME_WAIT period used by BSD TCP is 60 secs, the maximum connections in TIME_WAIT state at any instant is expected to be about $800 * 60 = 48000$. However, the results with conventional benchmark in Figure 3 indicate only a maximum of about 12000 connections in TIME_WAIT. We attribute this to a fallacy arising due to the limited number of client machines used in the experiment and is explained below.

A TCP connection on a machine is uniquely identified by the 4-tuple *(source IP address, destination IP address, source port number, destination port number)*. The port numbers are 16-bit entities and all connections from a client to the webserver are distinguished by the client port number. When a client is repeatedly initiating connections to the webserver at a high-rate, the port numbers can wrap around while an old connection having the same 4-tuple is still in the TIME_WAIT state at the server. Upon reception of a request for such a new connection, the TCP at the webserver typically terminates the old connection in the TIME_WAIT state. A real webserver receives requests from thousands of clients and very seldom does a client port number wrap around while an earlier connection with the same port number from the same client is still in the TIME_WAIT state. Thus, in an experimental setup with a limited number of client machines, the number of connections in TIME_WAIT can be far less than in an actual webserver.

We next propose a technique that enables webserver benchmarks to more accurately predict the performance of real servers.

	Conventional Benchmark	Modified Benchmark
Apache throughput (conn/s)	801	667
Max connections in TIME_WAIT	11928	48048
Max time in fasttimeo (ms)	6	24
Max time in slowtimeo (ms)	15	63
Max CPU Ovhd (%)	5.8	25
Max Kernel Dynamic Memory (MB)	9	32

Fig. 3. **Overhead of TCP timers**

A.1 Modification to Client Benchmark

We have shown that conventional benchmark environments consisting of a limited number of clients cause a premature termination of connections in TIME_WAIT state at the server and thus fail to fully expose the CPU and memory overheads. One way to fix this problem would be to modify the kernel at the webserver machine for the purpose of the experiment so as not to discard such connections prematurely. However, we were interested in a simpler solution that could be applied in environments where kernel modifications are not feasible.

We now propose a technique that can be applied to client benchmark machines for the purpose of measuring webserver performance representative of real servers. Our methodology is also applicable to other operating systems and for measuring the capacity of faster servers, whether based on a single machine (like Flash[14] and Zeus[15]) or on a cluster (like IBM's Netdispatcher[16] and LARD[17, 18]). The key for making this possible is to realize that the 4-tuple *(client IP address, server IP address, client port number, server port number)* should not repeat itself for a period of time as long as the wait in the TIME_WAIT period.

For this purpose, we changed our clients as follows. We aliased N IP addresses[2] on every client machine's network interface. The client applications used these IP addresses in a round-robin fashion while initiating connections to the server. The number N was chosen such that

$$N \geq C * T / P$$

where C is the maximum rate at which connections are initiated by individual client machines, T is the TIME_WAIT period (60 secs for BSD TCP) and P is the size of the port number space made available by the operating system to the client applications[3]. Intuitively, this causes an N-fold increase in the space of 4-tuples where a tuple gets repeated after an amount of time larger than the TIME_WAIT period.

[2] Most modern operating systems allow the system administrator to assign several IP addresses to individual network interfaces on the machine.

[3] Although the port number is 16 bits, not all 65536 ports are made available by the operating system for unprivileged user applications.

For our experiments, we used a value of 300 conn/s for C, 60 secs for T and 4000 for P (BSD allocates port numbers ranging from 1024 to 5000 for user applications). This gives a value of 5 for N and the results with the modified benchmark are shown in Figure 3.

Two key results can be observed in Figure 3. First, the CPU overhead due to the fasttimeo and slowtimeo functions can be as large as 25% and is principally caused by the periodic linear scans made over the large number of connections in TIME_WAIT. Second, the peak kernel dynamic memory utilization can be as large as 32 MB. This is a significant fraction of physical memory available on current off the shelf hardware and amounts to one fourth the memory on our webserver machine. Our results indicate that most of the memory overhead is also caused by the large number of connections in the TIME_WAIT state. Moreover, this overhead is expected to increase in the future as the number of connections in TIME_WAIT increases with higher throughput and with higher CPU speeds. This is discussed further in Section V where we also propose implementation enhancements that reduce both the CPU and memory overhead associated with the TIME_WAIT state.

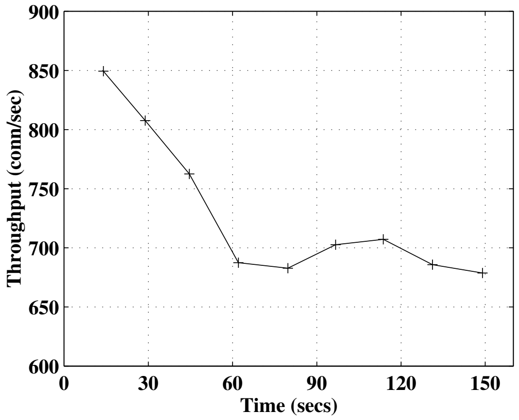

Fig. 4. **Apache startup**

Our results also lead us to make another observation with respect to webserver benchmarking. Figure 4 shows the throughput variation of the Apache webserver upon startup with the passage of time. The results show that the throughput decreases significantly (up to 25%) over the first 60 seconds before the performance becomes stable. The reason for this degradation is again caused by the wait in the TIME_WAIT state. During this period, no connections get terminated and therefore, the TCP connection termination overhead doesn't affect the throughput. Second, due to the continuous connection buildup, the overhead of maintaining the connections (i.e. the linear scans in fasttimeo and slowtimeo in BSD) worsens over the first 60 seconds and becomes stable only once connections start getting terminated.

We conclude that a webserver benchmark should at least wait for the TIME_WAIT period (60 secs for BSD) before it starts recording any results. Although intuitive, we feel that the issue of stable webserver startup behavior is not widely understood. While SPECweb96[19] does employ a period of 5 minutes to allow the server to reach steady state, some other popular webserver benchmarks like Webstone[20] start recording results without waiting for the minimum stabilization period.

B. Timing problems due to coarse-grained clock

Reliable transport protocols for lossy networks like TCP must ultimately rely on retransmission timeouts (RTO) to recover from packet losses. These timeouts must be conservative to prevent unnecessary retransmissions. However, overly conservative timeouts cause unnecessarily long transmission stalls in the event of losses, and may result in poor response time, low network utilization and throughput loss. This problem is increasingly costly as the bandwidth-delay product of wide-area networks increases.

TCP designers have long recognized this problem. TCP's fast retransmit [5] mechanism attempts to recover from packet losses without requiring a retransmission timeout. However, with multiple packet losses or when the send window is small, TCP still has to rely on a retransmission timeout for recovery. Despite the conventional wisdom of relying less on timeout based mechanisms [21], recent studies [22, 23, 24] have indicated that almost 50% of lost packets in the Internet are recovered by retransmission timeouts.

The BSD implementation of TCP uses a coarse-grained clock (500 msecs granularity), both for measuring the RTT (round-trip time) and for scheduling timeouts. The slowtimeo handler (see Section II-A) checks for pending timeouts regularly every 500 msecs rather than being synchronized every time an event is scheduled. For this reason, the minimum possible timeout value is 2 ticks even when the network round-trip time is significantly smaller than 500 msecs. As shown in Figure 5, a value of 2 ticks can result in an actual timeout between 0.5–1 secs. In general, if the granularity of event scheduling is D, and the desired timeout is T ticks, then the actual timeout can vary between $(T-1)*D$ and $T*D$.

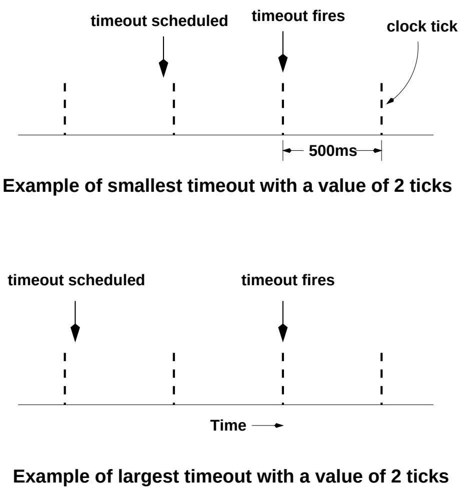

Fig. 5. **Timeout variation with value of 2 ticks**

Furthermore, Brakmo and Peterson [8] have observed that even for RTTs as small as 100 msecs, the timeout estimation algorithms of TCP can cause the value of the timeout to be as large as 5 ticks (causing the actual timeout to be between 2–2.5 secs) and almost never goes below 3 ticks. They also observe that the problem is caused due to the usage of the coarse-grained (500 msecs) clock to measure the round-trip time, resulting in numerical errors due to limited precision. Appendix I discusses this in more detail. Brakmo and Peterson also propose fixes in the form of larger scaling factors for timeout estimation equations that reduce the numerical errors associated with limited precision.

Some operating systems use finer clock granularities in their TCP implementations than the 500 msecs used by the BSD-based systems. Linux [25] uses a clock with 1 msec granularity for both measurement as well as scheduling. While we expect that the timeout delays in Linux are not as conservative as in BSD-based systems, the use of 1 msec clock might still be too coarse for measurements on high-bandwidth networks. Paxson [22] observes that on networks with a packet size of 512 bytes, the packet-pair algorithm [6] cannot distinguish between bandwidths higher than 4 Mbps with a clock granularity of 1 msec.

We propose a timer framework in Section VI that allows portable network protocol implementations to take advantage of the best facilities for interval measurement and event scheduling available in a given operating system. Given a desired timeout of T, it permits the actual timeout to be between T and $T+D$ where D is the granularity of event scheduling. Additionally, it permits the use of cycle counters available on most modern architectures that enable time measurement at very fine granularity (micro to nano second). These in turn can be used to measure high bandwidths.

V. TCP TIME_WAIT PROCESSING

In Section IV, we showed that there are significant CPU and memory overheads associated with TCP TIME_WAIT processing. This section proposes implementation techniques in TCP that reduce these overheads. We first present a timer implementation and show that it can avoid the CPU overhead associated with TIME_WAIT processing. Then, we present an implementation technique that reduces the memory overhead. Our techniques require no change to the TCP protocol or the API. For this reason, these techniques only require implementation enhancements to the kernel on the webserver machine.

A. Reducing TIME_WAIT CPU overhead

In this subsection we present an implementation that dramatically reduces the overhead associated with TIME_WAIT processing in TCP.

The implementation of timers in TCP involves a tradeoff. In BSD TCP, the setting and cancellation of timers is optimized at the expense of checking and firing a timeout. Thus, while setting and canceling a timer only requires setting a corresponding integer variable (as explained in Section II-A), the check to determine the execution of a timeout handler involves a linear scan over all the connections. This tradeoff pays well for active connections because in the common case timers are usually set and canceled several times before a linear scan to check for a timeout. However, the timer for terminating connections in TIME_WAIT is rarely canceled once it is set. Moreover, since the TIME_WAIT period is significantly larger than the period of the linear scans, each connection in the TIME_WAIT state undergoes many unnecessary checks before the handler for the TIME_WAIT timer is executed.

We propose an implementation that preserves the fast setting and cancellation of timers in BSD, but avoids the inefficiency of unnecessarily scanning connections in TIME_WAIT state. This is achieved by organizing the connections in the linear list scanned by the slowtimeo and fasttimeo functions such that connections in TIME_WAIT state are only scanned if needed.

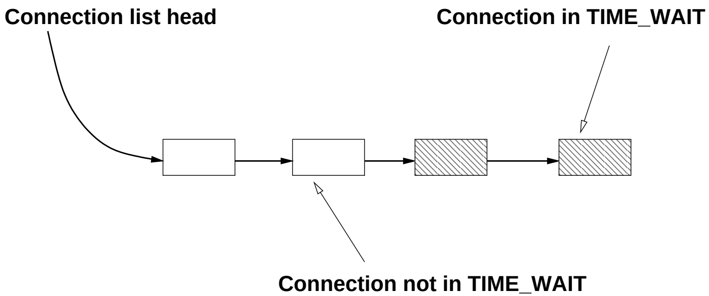

Fig. 6. **Efficient list-based timer implementation**

Our proposed implementation is depicted in Figure 6. Connections are moved to the end of the connection list as soon as they enter (or restart) the TIME_WAIT state. This automatically ensures that all connections in TIME_WAIT are sorted in increasing order of the time when the handler for the TIME_WAIT timer needs to be executed. The fasttimeo function was modified to stop scanning the list as soon as it reaches the first connection in the TIME_WAIT state (since connections in TIME_WAIT do not delay ACKs[4]). The slowtimeo function was modified to stop scanning the list as soon as it discovers a connection in TIME_WAIT whose handler only needs to be executed in the future (since connections in TIME_WAIT are sorted, this ensures the invariant for the rest of the connections too).

	Original	New
Apache throughput (conn/s)	667	834
Max time in fasttimeo (ms)	24	0.4
Max time in slowtimeo (ms)	63	0.8
Max CPU Ovhd (%)	25	0.36

Fig. 7. **Comparison of Overheads**

The results for this improved list-based implementation are shown under the column labeled "New" in Figure 7. For comparison, earlier results quantifying the overheads in BSD TCP are also shown under the column labeled "Original". The numbers indicate that with the new implementation, the overhead of the TCP timers becomes negligible and the available CPU cycles are converted into a throughput improvement of 25% by the Apache webserver.

For the purpose of comparison, we also implemented a timing wheel[26] based approach for the management of TCP timers. Our results indicate that our list-based implementation outperforms the timing wheel implementation by about 5%. This is because the setting and cancellation of events in a timing wheel involves pointer manipulations and the computation of a hash function, which is more expensive than simply setting an integer variable.

A concern with our list-based implementation is that it still might impose unnecessary scanning overhead with idle connections resulting from the use of HTTP/1.1 persistent[12, 2] connections. For this purpose, we repeated the above experiment after 10000 idle connections were established between the webserver and the client machines. Our results indicate that even with this relatively large number of idle connections, performance only degraded by about 5%. Moreover, the overhead imposed by idle connections can be limited by the webserver applica-

[4] ACKs are only delayed in the ESTABLISHED[5] TCP state.

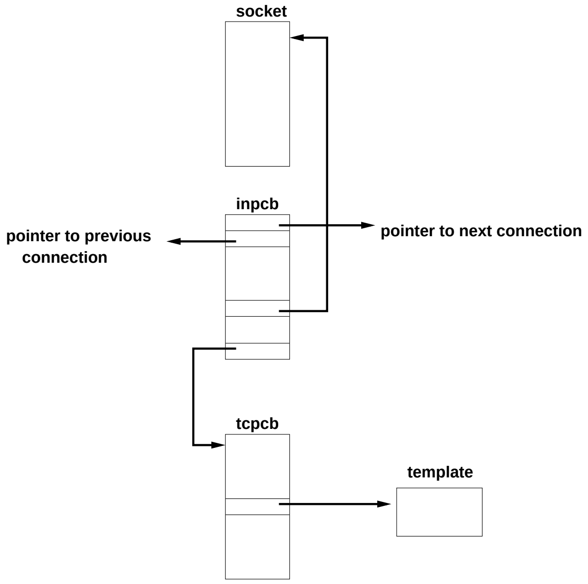

Fig. 8. **Kernel Data Structures**

tion by restricting the maximum number of such connections.

B. Memory loading due to TIME_WAIT

In Section IV-A.1 we quantified a significant memory overhead (up to 32 MB) associated with the kernel in a busy webserver. In this subsection, we discuss reasons for this overhead and show that the overhead is primarily caused by the large number of connections in TIME_WAIT. We also propose implementation changes that immensely reduce this overhead.

Figure 8 shows the per-connection data structures maintained by the BSD kernel. All connections are maintained in a doubly linked list of Internet protocol control blocks (inpcb) – this is the list that is scanned by the slowtimeo and fasttimeo functions. A connection's inpcb contains the source and destination IP addresses and port numbers, cached routing information, pointers for list maintenance etc. Also associated with a TCP connection is another control block that maintains TCP specific state (tcpcb) such as congestion control and flow control information, TCP timers, a template that caches header information regarding outgoing packets, etc. A socket structure is also maintained that contains such state as the send and receive buffers, information regarding the process that owns the socket etc. In all, about 600 bytes of kernel state is maintained for each connection and occupies the physical memory of the machine.

In Section IV-A.1 we showed that the number of connections in TIME_WAIT state can be as large as 48000 for the Apache webserver. Maintaining large amounts connection state for such inactive connections can cause significant memory overhead. Moreover, kernel dynamic memory is not pageable, and thus occupies physical memory on the machine.

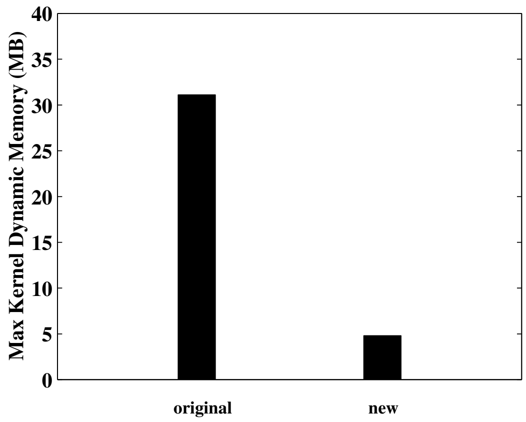

Fig. 9. **Kernel Memory at Webserver**

To reduce the memory overhead caused by the connections in the TIME_WAIT state, we modified the kernel so as to discard all the state depicted in Figure 8 as soon as a connection enters TIME_WAIT. Instead, a data structure carrying a small amount of state (40 bytes) was inserted in the connection list. This state was sufficient for operation in the TIME_WAIT state and includes the source and destination IP addresses and port numbers, the latest TCP sequence numbers exchanged, the TIME_WAIT timer and some pointers for list manipulation.

The bars labeled "original" and "new" in Figure 9 compare the memory overheads between the BSD implementations with and without our proposed implementation technique, respectively. The results show that our implementation reduces the memory overhead by more than a factor of 6. The resulting peak consumption of about 5 MB can be easily afforded on off the shelf hardware. The physical memory made available to the webserver can be utilized for caching frequently requested documents in memory that can help improve throughput and user response time.

	Original	New
Apache throughput (conn/s)	216	379
Max Kernel Dynamic Memory (MB)	17	4

Fig. 10. **Performance with Real Trace**

To quantify the effect of improved memory utilization on throughput, we repeated the experiment with a workload trace obtained from Rice Computer Science department's webserver. The trace covered a period of 3 days and accessed over 3000 distinct files having an aggregate size of 87 MB. Figure 10 shows that with this workload, the throughput is 75% more with our proposed changes than without. The improvement is primarily attributed to the fewer disk accesses needed with our implementation as a result of a larger effective memory cache. The memory overheads reported in Figure 10 are lower than those reported in Figure 9 because the lower throughput resulting from frequent disk accesses causes fewer connections to be in the TIME_WAIT state simultaneously.

VI. Decoupling Time Measurement and Event Scheduling

In Section IV-B we described the limitations arising due to the use of a coarse-grained clock for time measurement and event scheduling in TCP. In this section, we propose a timer framework that decouples the time measurement facility available to the operating system from the event scheduling timer facility, thus enabling best use of the system capabilities. We then present experimental results showing that our framework is capable of dramatically reducing the time taken by retransmission timeouts (RTO) in TCP.

A. Timer Framework

We describe an OS timer framework that allows network protocol implementations and other system components to make use of the timer facilities available in the host operating system and architecture in a portable way.

The framework provides separate facilities for the measurement of intervals and the scheduling of events. By separating the facilities for interval measurement and event scheduling and by allowing protocol implementations to query the granularity of each facility, protocol implementations can make the best use of the available timer support.

Our proposed framework has two salient features:

- Clients (e.g., network protocol implementations) expect the OS to provide two separate timer-related services—one for measurement and one for event scheduling.
- The clients are aware of the clock granularity of each of the services and can adapt their behavior accordingly.

Typically, the OS service for measuring intervals can provide a finer granularity than the service for event scheduling. Providing an event timer involves a periodic interrupt that is currently only efficiently supported at millisecond granularity, because a finer granularity could cause significant overhead. On the other hand, most modern architectures provide a fine-grained timer/counter with micro to nano-second granularity in hardware, whose current value can be read with very little overhead. The separation of the measurement and event scheduling services in our framework allows a protocol implementation to take advantage of this feature.

The OS service for interval measurement consists of the operations `measure_time()` and `measure_resolution()`. The former returns a 64-bit value representing the current real time in ticks of a clock whose resolution is given by the latter operation (in Hz). Since these operations are used to measure time intervals, the time need not be synchronized with any standard time base.

A second OS service provides for a facility to schedule events. This conventional facility is implemented in the OS by a periodic timer interrupt. This service provides the following three operations:

1. `schedule_event(T, handler)`: schedules the function *handler* to be called at least T ticks in the future (the resolution of T is specified by `measure_resolution()`). We say that T is the required time of the event.
2. `schedule_resolution()`: gives the expected resolution (in Hz) at which the OS can schedule events and equals the frequency of the periodic timer interrupts used by the OS to implement the service.
3. `cancel_event(event)`: provides for the cancellation of a previously scheduled event (the argument *event* is returned by `schedule_event()`).

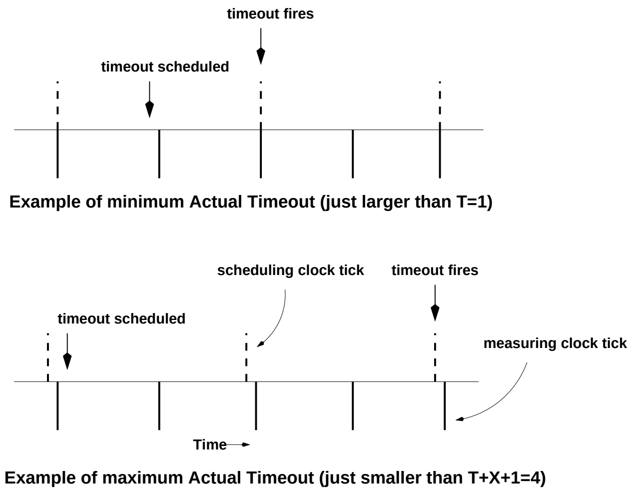

Fig. 11. **Lower and Upper Bounds for Actual Timeout**

The timer facility fires an event (by calling the corresponding handler) when the value returned by `measure_time()` exceeds the value stored at the time the event was scheduled by at least $T + 1$ (the increment by one accounts for the fact that the time at which the event was scheduled may not exactly coincide with a timer interrupt). If X is the resolution of the scheduling clock relative to the measurement clock (i.e., $X \equiv measure\_resolution()/schedule\_resolution()$), then our framework puts the following bounds on the actual time (in ticks of the measurement clock) when the event fires:

$$T < Actual\ Event\ Time < T + X + 1$$

RTT_{coarse} (500ms)	$ERTO_{Orig}$ (500ms)	RTO_{Orig} (1ms)	$ERTO_{Brakmo}$ (500ms)	RTO_{Brakmo} (1ms)	RTT_{fine} (1ms)	$ERTO_{New}$ (1ms)	RTO_{New} (1ms)
0	3	1375	3	1375	60	183	375
0	3	1314	3	1314	150	253	314
1	5	2254	3	1254	60	218	254
0	4	1193	3	1193	60	190	193
0	3	1133	3	1133	61	167	633
0	3	1073	3	1073	60	150	573
0	3	1012	3	1012	61	135	512
0	3	1452	3	1452	60	123	452
0	3	1391	3	1391	60	114	391
0	3	1330	2	830	61	105	330
0	3	1269	2	769	60	99	269
1	5	2209	3	1209	61	93	145

Fig. 12. **Sample RTO computation**

Figure 11 gives examples of the above bounds when $T = 1$ and $X = 2$. It is to be noted that the increment by one in the upper bound is negligible if the measurement clock is significantly finer than the scheduling clock (as is the case in most modern systems).

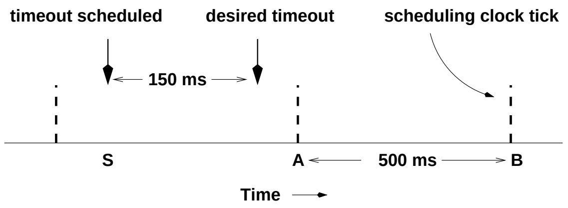

Fig. 13. **Example usage with TCP**

Figure 13 shows an example where our framework achieves a lower delay in the retransmission timeout for TCP. Assume for the sake of argument that the time measurement clock has a granularity of 1 msec (thus $X = 500$). A timeout is scheduled at time S for $T = 150$ msec. As timeouts are only checked at scheduling clock ticks every 500 msec (in slowtimeo), our framework would invoke the handler for the timeout at time depicted in the figure by A. This is because our framework will maintain all values at a granularity of 1 msec, thus permitting the check $A - S \geq T + 1$ to succeed.

The original BSD implementation would only invoke the handler at or later than time B because it imposes a minimum of 2 clock ticks on the timeout (as explained in Section IV-B). Moreover, the coarse-grained measurement of RTT can cause the timeout value to be larger than 2 ticks as observed by Brakmo and Peterson[8]. It is interesting to observe that if the desired timeout value is small in comparison to 500 msec, the conventional BSD TCP implementation puts a *lower bound* of 500 msec on the timeout while our framework puts an *upper bound* of 500 msec.

In addition to reducing the delays associated with a timeout, the ability to make fine-grained interval measurements can be employed for other protocol functionality, for instance to measure high bandwidths using the packet-pair algorithm.

B. Experimental Results

In this subsection, we show the use of our proposed timer framework in achieving less conservative retransmission timeouts in TCP. Specifically, we make use of the ability to measure intervals at fine grain to obtain better RTT (round-trip time) estimates, and we use the conventional timer facility to schedule retransmission timeouts at a granularity of 500 msec. The original TCP algorithm, as described in Appendix I, was used to make the retransmission timeout (RTO) estimation – however, the computation was performed at the granularity of the fine-grained measurement clock.

To produce realistic TCP timeout estimates, we used the experimental topology of Figure 2 where the delay router was used to emulate a round-trip propagation delay of 60 msec. The topology resembles a real internetwork where data is transferred from the server across a WAN link to hosts on several 100Mbps LANs.

Figure 12 shows a series of measured RTT values for a typical server connection, along with the resulting estimated RTO values taken from a typical TCP sender in the topology. RTT_{coarse} gives a series of successive RTT measurements made by a 500 msecs clock, RTT_{fine} gives the same as measured by the Pentium cycle counter but reported at 1 msec granularity for readability. Upon every new RTT measurement, TCP estimates the desired RTO value and resets the retransmission timer with this value. $ERTO_{Orig}$ gives the RTO estimate (the desired value of timeout) as computed by the BSD TCP implementation, $ERTO_{Brakmo}$ gives the same after applying the fixes suggested by Brakmo and Peterson [8] to the timeout estimation algorithms. The columns labeled RTO_{Orig} and RTO_{Brakmo} denote the actual time (in msecs) that a timeout would have taken to fire without and with the Brakmo/Peterson fixes, respectively, assuming that the periodic timer interrupts for firing timeouts still happen at 500 msecs intervals. These are measured by computing the time difference between the instance when the handler would have been invoked and when the timeout was scheduled.

$ERTO_{New}$ gives the estimated value of timeout when the estimation algorithms use the finer-grained measurement clock using the timer framework presented in Section VI. Again we report the value at a millisecond granularity for readability. RTO_{New} is the actual time (in msecs) that a timeout would have taken to fire if it had been scheduled using the value given by $ERTO_{New}$[5].

A comparison of columns RTO_{Orig}, RTO_{Brakmo} and RTO_{New} shows that our framework is capable of achieving up to 15 and 8 times tighter timeouts as compared to the BSD implementation, without and with, respectively, the fixes proposed by Brakmo and Peterson.

As the bulk of HTTP traffic consists of relatively short TCP data transfers[12, 13], packet losses are often recovered by a timeout. A reduction in the delay associated with TCP timeouts thus directly translates into a corresponding reduction in response time of a webserver. Moreover, tighter delays associated with a timeout reduce underutilization of link bandwidths and a corresponding degradation in throughput.

VII. Related Work

Faber et. al. [27] discuss the overloading of busy web servers with TIME_WAIT state and propose to change the TCP protocol in order to shift the TIME_WAIT state to clients. Changes to the TCP protocol specification require support both at the client as well as the webserver machines. Our work aims to reduce the CPU and memory overhead associated with the management of connections in TIME_WAIT state and only requires implementation changes to the TCP at the webserver machine. No protocol or API changes are required.

Keshav[6] proposed the packet-pair technique that estimates the network bandwidth by observing the spacing between closely spaced packets. Hoe[7] proposes to set the TCP slow-start threshold to an appropriate value by measuring the bandwidth-delay product using a variant of the packet-pair technique. Padmanabhan et. al. [28] propose to use the packet-pair technique followed by rate-based pacing to alleviate the slow-start delays associated with short TCP transfers. We propose a timer framework that decouples the time measurement clock from the event scheduling clock, thus permitting time measurements to be made at a much finer granularity (micro to nano second if cycle counters are used). This enables the use of the packet-pair technique to measure high bandwidths. Paxson [22] observes that on networks with a packet size of 512 bytes, the packet-pair algorithm [6] cannot distinguish between bandwidths higher than 4 Mbps with a clock granularity of 1 msec. He also suggests a more robust bandwidth estimation technique called PBM that forms estimates using a range of packet bunch sizes.

Paxson[22] shows that in the absence of the SACK TCP option [29], a significant number of lost packets in the Internet are recovered using the coarse-grained retransmission timeout. The same was confirmed by Balakrishnan et. al. [23]. Our proposed timer framework reduces the unnecessary long delays associated with a retransmission timeout when a coarse-grained clock is used for scheduling events. Brakmo et. al. [30] have also used the fine-grained system clock to detect lost packets early (while processing a received duplicate ACK) in TCP Vegas. Their method does not however affect the coarse-grained retransmission timeout in TCP.

VIII. Conclusions

This paper makes several contributions. We show that the lack of scalability with respect to management of connections in the TIME_WAIT state imposes CPU and memory overheads up to 25% and 32 MB respectively. We propose efficient implementation techniques for reducing these overheads. Our proposed techniques only require implementation changes to the webserver machine and require no changes to the TCP protocol or the API. We also show that conventional benchmark environments do not fully expose the CPU and memory overheads associated with a busy webserver and propose benchmark techniques that allow a more accurate prediction of the performance of real servers.

We also propose a timer framework that decouples the time measurement clock from the event scheduling clock. This permits time measurement at very fine granularity by using cycle counters available with most modern machines. We present results that show that our framework is capable of dramatically (up to a factor of 15) reducing the delay associated with TCP retransmission timeout. In addition, the finer granularity afforded by the time measurement clock can be used to measure bandwidth accurately on high bandwidth networks using the packet-pair[6] algo-

[5]In practice, the minimum value allowable for the estimated value of RTO is 200 msecs because a TCP receiver can delay ACKs by that time.

rithm.

REFERENCES

[1] T. Berners-Lee, R. Fielding, and H. Frystyk, "RFC 1945: Hypertext transfer protocol – HTTP/1.0," May 1996.

[2] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee, "RFC 2068: Hypertext transfer protocol – HTTP/1.1," Jan. 1997.

[3] J. Postel, "RFC 793: Transmission control protocol," Sept. 1981.

[4] G.R. Wright and W.R. Stevens, *TCP/IP Illustrated Volume 2 : The Implementation*, Addison-Wesley, Reading, MA, 1995.

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

[6] S. Keshav, "A Control-Theoretic Approach to Flow Control," in *Proceedings of the ACM SIGCOMM '91 Symposium*, Sept. 1991, pp. 3–15.

[7] Janey C. Hoe, "Improving the Start-up Behaviour of a Congestion Control Scheme for TCP," in *Proceedings of the ACM SIGCOMM '96 Symposium*, 1996.

[8] Lawrence Brakmo and Larry Peterson, "Performance Problems in 4.4BSD TCP," *ACM Computer Communication Review*, vol. 25, no. 5, pp. 69–86, Oct. 1995.

[9] "Apache," http://www.apache.org/.

[10] Gaurav Banga and Peter Druschel, "Measuring the capacity of a Web server under realistic loads," *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 1999.

[11] Lixia Zhang, Scott Shenker, and David D. Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic," in *Proceedings of the ACM SIGCOMM '91 Symposium*, 1991, pp. 133–148.

[12] Jeffrey C. Mogul, "The Case for Persistent-Connection HTTP," in *Proceedings of the ACM SIGCOMM '95 Symposium*, 1995.

[13] Martin F. Arlitt and Carey L. Williamson, "Web Server Workload Characterization: The Search for Invariants," in *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

[14] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, "Flash: An efficient and portable Web server," in *Proceedings of the Usenix 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[15] "Zeus," http://www.zeus.co.uk/.

[16] IBM Corporation, "IBM interactive network dispatcher," http://www.ics.raleigh.ibm.com/ics/isslearn.htm.

[17] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum, "Locality-Aware Request Distribution in Cluster-based Network Servers," in *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[18] Mohit Aron, Peter Druschel, and Willy Zwaenepoel, "Efficient Support for P-HTTP in Cluster-based Web Servers," in *Proceedings of the Usenix 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[19] "SPECWeb96," http://www.specbench.org/osg/web96/.

[20] "WebStone," http://www.mindcraft.com/webstone/.

[21] Lixia Zhang, "Why TCP Timers Don't Work Well," in *Proceedings of the ACM SIGCOMM '86 Symposium*, Aug. 1986.

[22] Vern Paxson, "End-to-End Internet Packet Dynamics," in *Proceedings of the ACM SIGCOMM '97 Symposium*, 1997.

[23] Hari Balakrishnan, Srinivasan Seshan, Mark Stemm, and Randy H. Katz, "Analyzing Stability in Wide-Area Network Performance," in *Proceedings of the ACM SIGMETRICS '97 Conference*, 1997.

[24] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz, "TCP behavior of a busy internet server: Analysis and improvements," in *Proceedings of IEEE INFOCOM '98*, San Francisco, California, Apr. 1998.

[25] "Linux," http://www.linux.org/.

[26] G. Varghese and A. Lauck, "Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Nov. 1987, pp. 171–180.

[27] Theodore Faber, Joe Touch, and Wei Yue, "The TIME-WAIT state in TCP and its Effect on Busy Servers," in *Proceedings of IEEE INFOCOM '99*, Mar. 1999.

[28] Venkata N. Padmanabhan and Randy H. Katz, "TCP Fast Start: A Technique For Speeding Up Web Transfers," in *Proceedings of the IEEE GLOBECOM '98 Conference*, Nov. 1998.

[29] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "RFC 2018: TCP selective acknowledgment options," Oct. 1996.

[30] Lawrence Brakmo and Larry Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1465–1480, Oct. 1995.

[31] Van Jacobson, "Congestion Avoidance and Control," in *Proceedings of the ACM SIGCOMM '88 Symposium*, Aug. 1988, pp. 314–32.

APPENDIX

I. TCP TIMEOUT ESTIMATION

The RTO value calculation in the BSD implementations of TCP are described in [31]. The appropriate retransmission timer value rto is calculated based on an RTT estimator a and a mean deviation estimator d, according to the formula:

$$rto \leftarrow a + 4d$$

The estimators are updated using a new RTT measurement m using the following formulas:

$$Err \equiv m - a$$

$$a \leftarrow a + g_0 Err$$

$$d \leftarrow d + g_1(|Err| - d)$$

The chosen values for gain parameters are $g_0 = \frac{1}{8}$ and $g_1 = \frac{1}{4}$, which allows integer arithmetic using scaled versions of the above equations and using scaled estimators that are defined as $sa = 2^3 a$ and $sd = 2^2 d$. The calculations can be implemented using integer arithmetic as follows:

$$Err \equiv m - (sa >> 3)$$

$$sa \leftarrow sa + Err$$

$$sd \leftarrow sd + (|Err| - (sd >> 2))$$

$$rto \leftarrow (sa >> 3) + sd \tag{1}$$

The BSD implementations of TCP use a coarse-grained clock (500ms granularity) for measuring RTT and scheduling timeouts. Two problems arise when the above calculations are performed on ticks from a coarse-grained clock:
1. Due to the coarse-grained clock used, there can be a large variance in the measurement of RTT, resulting in a

large value for sd. For small RTTs, the contribution of $(sa >> 3)$ would be much smaller than the contribution due to sd, thus resulting in large estimates of RTO as compared to the average RTT (given by $(sa >> 3)$).
2. Brakmo and Peterson [8] have shown that due to limited precision of the C implementation, the mean deviation estimator does not decay even when repeated RTT measurements with the same value are made. This problem is significant only when sd is large compared to $(sa >> 3)$ i.e. when the clock is coarse-grained.