The TIME-WAIT state in TCP and Its Effect on Busy Servers

Theodore Faber
Joe Touch
Wei Yue
University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
Phone: 310-822-1511
{faber,touch,wyue}@isi.edu

Abstract - **Hosts providing important network services such as HTTP and FTP incur a per-connection memory load from TCP that can adversely affect their connection rate and throughput. The memory requirement is directly tied to the number of connections; caching and other sharing methods will not alleviate it. We have observed HTTP throughput reductions of as much as 50% under SunOS 4.1.3 due to this loading.**

This paper advocates off-loading the memory requirements to the growing number of clients. This reduces server memory requirements as connection rate at that server grows due to increases in the number of clients and the bandwidth available on the network. Our approaches control server memory load better with growing client load than per-transaction techniques such as persistent HTTP connections. Our approaches also interoperate with persistent connections to take advantage of their other benefits.

This paper describes the causes of the memory loading, called *TIME-WAIT loading*, and defines three methods of alleviating it that scale with increasing number of clients. We present measurements of the systems and a comparison of their properties.

1. Introduction

The Transmission Control Protocol (TCP)[1] provides reliable byte-stream transport to hosts on the Internet. TCP is used by most network services that require reliable transport, including the Hypertext Transport Protocol (HTTP)[2]. TCP's method of isolating old connections from new ones results in an accumulation of state at busy servers that can reduce their throughput and connection rates. The effect on HTTP servers is of particular interest because they carry a large amount of Internet traffic.

TCP requires that the endpoint that closes a connection blocks further connections on the same host/port pair until there are no packets from that connection remaining in the network[2]. Under HTTP, this host is usually the server[2].

To temporarily block connections, one endpoint keeps a copy of the TCP control block (TCB) indicating that the connection has been terminated recently. Such a connection is in the `TIME-WAIT` state[1]. Connections in `TIME-WAIT` are moved to `CLOSED` and their TCB discarded after enough time has passed that all packets from the same connection have left the network. Packets leave the network by arrive at one of the endpoints and being rejected, or arriving with an expired time-to-live (TTL) field at a router and being deleted.

For endpoints that are the target of many connections, thousands of connections may be in `TIME-WAIT` state at any time, which introduces significant memory overhead. We refer to this condition as *TIME-WAIT loading*.

If the endpoint's TCP implementation searches all TCBs when delivering packets, `TIME-WAIT` loading will directly affect its performance. The presence of many `TIME-WAIT` TCBs can increases the demultiplexing time for active connections. We have seen throughput drop by 50% at loaded endpoints, and the effect on commercial servers has been noted elsewhere[3]. Some TCP implementations address the demultiplexing problem without addressing the memory load; we discuss them in Section 2.2.

The design of TCP places the `TIME-WAIT` TCB at the endpoint that closes the connection; this decision conflicts with the semantics of many application protocols. The File Transfer Protocol (FTP)[4] and HTTP both interpret the closing of the transport connection as an end-of-transaction marker. In each case, the application protocol requires that servers close the transport connection, and the transport protocol requires that servers incur a memory cost if they do. Protocols that use other methods of marking end-of-transaction, e.g., SUN RPC over TCP[5], can have the clients close connections at the expense of a more complex application protocol.

If the number of clients continues to increase, the only way to keep server `TIME-WAIT` memory requirements constant is to move the `TIME-WAIT` TCBs to clients. Aggregating more requests per connection merely reduces the growth of the memory load with respect to increasing client load; moving the load to clients distributes server memory load to the cause of that load.

As networks become faster and support more users, the connection rates at busy servers are likely to increase, resulting in more `TIME-WAIT` loading. Even if packet demultiplexing is done efficiently, the memory cost of `TIME-WAIT` loading can become a significant drain on server resources. Servers will need additional physical memory resources to support the load. For embedded servers using TCP, this translates directly to a higher cost in dollars and power.

Distributing the TCBs across clients scales better than per-transaction load reductions like persistent HTTP connections for controlling `TIME-WAIT` loading. The transaction rate is being driven up by the increasing bandwidth and the growing number of users. Reducing each transaction's cost only slows the growth rate. Offloading TCBs to clients distributes the load more equitably as the number of clients grows, riding the growth curve instead of throttling it. Because Persistent connections reduce other per-connection costs, such as extra connection establishment overhead, our systems interoperate with them.

This work presents three systems to distribute the `TIME-WAIT` build-up to clients. We suggest avoiding `TIME-WAIT` loading by negotiating which endpoint will hold the `TIME-WAIT` TCB during connection establishment. This provides the most control over `TIME-WAIT` location by making the placement an explicit part of connection establishment; however, performing this negotiation and respecting it when the connection is closed requires significant changes to TCP.

In light of this, we also discuss two less invasive alternative solutions: a modification to TCP that shifts the `TIME-WAIT` TCB from server to client when the connection is closed; and a modification to HTTP that supports the client closing the connection and holding the `TIME-WAIT` TCB.

2. The `TIME-WAIT` State

This Section discusses the `TIME-WAIT` state and its use in TCP in some detail, and how the `TIME-WAIT` state impacts the performance of busy servers.

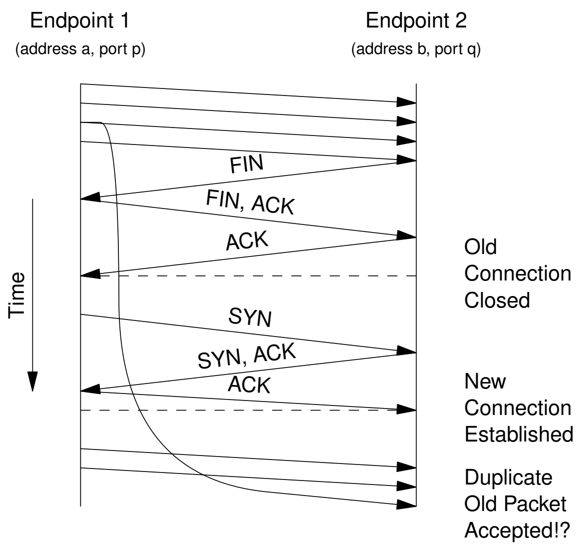

Figure 1: The Problem Addressed by the `TIME-WAIT` State

2.1. The Function of `TIME-WAIT`

The purpose of `TIME-WAIT` is to prevent delayed packets from one connection being accepted by a later connection. Concurrent connections are isolated by other mechanisms, primarily by addresses, ports, and sequence numbers[1].

The `TIME-WAIT` state avoids the situation depicted in Figure 1. Arrows represent packets, and endpoints' time lines run down the page. Packets are labelled with the header flags that are relevant to connection establishment and shutdown; unlabelled packets carry only data.

Specifically:

- A connection from (address a, port p) to (address b, port q) is terminated
- A second connection from (address a, port p) to (address b, port q) is established
- A duplicate packet from the first connection is delayed in the network and arrives at the second connection when its sequence number is in the second connection's window.

If such a packet appears, there is no way for the endpoints in the second connection to determine that the delayed packet contains data from the first connection.

This confusion can only exist if a second connection from (address a, port p) to (address b, port q) is active while duplicate packets from the first connection are still in the network. TCP avoids this condition by blocking any second connection between these address/port pairs until one can assume that all duplicates must have disappeared.

Connection blocking is implemented by holding a `TIME-WAIT` TCB at one endpoint and checking incoming connection requests to ensure that no new connection is established between the blocked addresses and ports. Because only a connection between the same endpoints can cause the confusion, only one endpoint needs to hold the state. The TCB is held for twice the maximum segment lifetime (MSL).

The MSL is defined as the longest period of time that a packet can remain undelivered in the network. Originally, the TTL field of an IP packet was the amount of time the packet could remain undelivered, but in practice the field has become a hop count[6]. Therefore, the MSL is an estimate rather than a guarantee. The Internet host requirements document suggests a using 2 minutes as the MSL[7], but some implementations use values as small as 30 seconds[8]. Under most conditions waiting $2 \times$ MSL is sufficient to drain duplicates, but they can and do arrive after that time. The chance of a duplicate arriving after $2 \times$ MSL is greater if MSL is smaller.

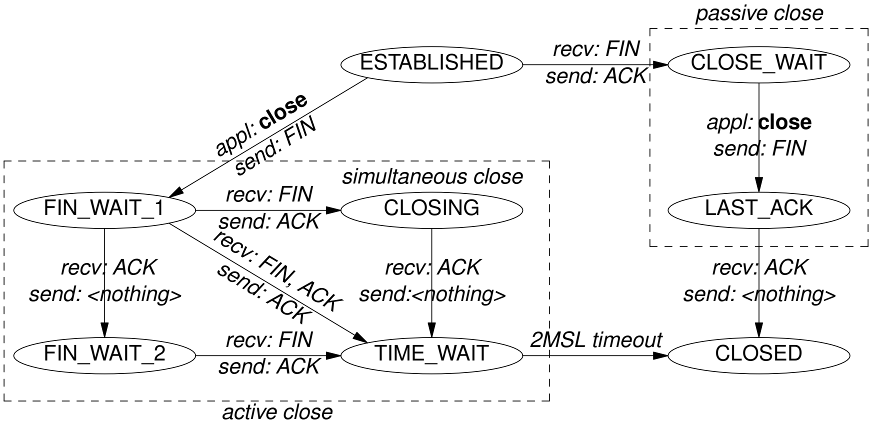

Figure 2: The TCP State Machine For Closing Connections[1]

TCP requires that the endpoint that initiates an active close of the connection eventually enters `TIME-WAIT`. Closing a connection that is in the `ESTABLISHED` state is called *actively closing*, closing from `CLOSE-WAIT` is *passively closing*. If both ends close the connection from `ESTABLISHED`, this is a simultaneous close, and both endpoints do a modified active close[1]. (See Figure 2.) Intuitively, the first endpoint to close a connection closes it actively, and the second passively; HTTP and FTP servers generally close connections actively.

Tying the `TIME-WAIT` state to the closing method simplifies the TCP state diagram because no information from the connection establishment affects connection termination.

2.2. Performance Problems at Busy Servers

Because client/server protocols are generally synchronized request/response protocols, the protocol specification usually determines which endpoint will close the transport connection. For example, FTP clients know a file has been delivered successfully if the connection on which the file was transferred closes gracefully[4]; this implies that the server closes connections.

TCP commentators encourage client/server systems to arrange for the client to close connections to avoid `TIME-WAIT` loading[8]. Many protocols, such as FTP, do not follow this convention. We discuss the reasons for this in Section 2.3.

Because application protocols do not take `TIME-WAIT` TCB distribution into account, heavily loaded servers can have thousands of connections in `TIME-WAIT` that consume memory and can slow active connections. In BSD-based TCP implementations, TCBs are kept in mbufs, the memory allocation unit of the networking subsystem[9]. There are a finite number of mbufs available in the system, and mbufs consumed by TCBs cannot be used for other purposes such as moving data. Some systems on high speed networks can run out of mbufs due to `TIME-WAIT` buildup under high connection load. A SPARCStation 20/71 under SunOS 4.1.3 on a 640 Mb/s Myrinet[10] cannot support more than 60 connections/sec because of this limit.

Demultiplexing incoming packets requires searching the endpoint's list of TCBs which will be full of of `TIME-WAIT` TCBs at a `TIME-WAIT` loaded server. In a simple implementation, the TCB list is searched linearly to pass the packet to the appropriate connection, which can be a bottleneck. The additional search overhead can cut throughput in half between two SunOS 4.1.3 SPARCStations on a Myrinet. We show an example in Section 5.

Modern TCP implementations avoid the overhead of the linear search of `TIME-WAIT` TCBs when demultiplexing packets. BSDI/OS keeps `TIME-WAIT` TCBs at the end of the list of TCBs, so that they can be used as a terminator for the linear search[11]. Looking TCBs up in a hash table reduces lookup times both for systems with many `TIME-WAIT` TCBs and for many active connections[12].

Some systems address `TIME-WAIT` loading by using a shorter MSL, hoping to age the `TIME-WAIT` TCBs out of the system sooner, which weakens the protection afforded by `TIME-WAIT`. If `TIME-WAIT` TCBs are kept at clients, they can afford to keep them for the full MSL. Using a shorter MSL at servers alleviates the memory usage problem, but can affect the integrity of communications with any host to which it connects. The size of the MSL to maintain a given memory usage level is inversely proportional to the connection rate. This implies that connections will be least protected at the most loaded servers.

Even in TCP implementations that demultiplex packets efficiently, such as those mentioned above, `TIME-WAIT` TCB accumulation consumes memory. Systems such as persistent HTTP connections can reduce the per-transaction server memory cost, but the Internet continues to grow both in terms of available bandwidth and in terms of number of users. The result is that busy servers will be seeing more

[1] The diagram layout is modelled after one appearing in [8].

connections per unit time, which translates directly into increased memory usage.

Spreading the memory requirement to the growing number of clients scales better than reducing the per-transaction cost at the server when the number of transactions is still growing. Per-transaction systems try to reduce the cost of each client, but the requirements still grow with increasing Internet traffic. Distributing the `TIME-WAIT` TCBs to clients takes advantage of their growing number to reduce the memory load on servers. Our systems support persistent connections in order to share their other benefits, such as avoiding extra TCP 3-way handshakes.

2.3. Server Close and Application Semantics

Using connection close to delimit transactions is a clean abstraction with unintended performance impact under TCP. HTTP and FTP servers signal the end of a transaction by closing the transport connection. As we argued above, this can `TIME-WAIT` load servers[2,4]. These protocols carry the majority of the TCP traffic in the wide area Internet today.

Using the TCP connection closure as part of a simple request/response protocol results in simpler protocols. In such as system, the semantics of TCP's close make it an unambiguous end-of-transaction marker.[2] Without this mechanism, protocols are forced to specify transaction length explicitly, or to provide an unambiguous end-of-transaction marker.

Providing an unambiguous end-of-transaction marker in the data stream requires that the server either knows the content length when the response begins, or edits the outgoing data stream to mask end-of-transaction markers in the data that the client must restore. Transactions must have a length field or be byte-stuffed.

If a response is generated dynamically, its length may not be known when the response starts, making the former framing methods unwieldy. Such data may be generated on demand from another program in the system that does not respect the protocol end-of-file marker. An HTTP server that supports the Common Gateway Interface (CGI) is such a system. Buffering that program's output to determine its size or remove embedded end-of-transaction markers slows the response.

3. `TIME-WAIT` Negotiation

This section discusses modifying TCP to negotiate the `TIME-WAIT` TCB holder when the connection is established. This makes the post-connection memory requirement explicit and allows either endpoint to decline the connection if the overhead is unacceptable. Furthermore it is transparent to applications using the transport, and allows them to close the transport connection as part of their protocol without incurring hidden costs.

[2] This is false in protocols that can have multiple pending requests, e.g., pipelined HTTP requests[13].

We propose adding a TCP option, TW-Negotiate, that indicates which end of the connection will hold the TCBs. TW-Negotiate will be negotiated during the three-way handshake that is used to synchronize TCP sequence numbers. The three-way handshake has been used to negotiate other options, such as TCP window scaling[14].

The negotiation algorithm for a client/server connection:

1. Client includes the TW-Negotiate option in the `<SYN>` packet for the connection. TW-Negotiate contains the IP address of the end that will hold the TCB. The option's presence indicates that the client supports negotiation. Clients must send an IP address, to support the algorithm below for resolving a simultaneous open.
2. Server returns the `<SYN, ACK>` packet with TW-Negotiate set to its choice to keep the `TIME-WAIT` state. If it does not support negotiation, it sends no TW-Negotiate option.
3. The client decides if the server's choice is acceptable. If so, it acknowledges the `<SYN, ACK>` packet with the same value of TW-Negotiate. If not it aborts the connection with an `<RST>` packet. (The connection is aborted as though it failed to synchronize, and introduces no new failure modes to TCP.) Aborting the connection from this unsynchronized condition leaves no extra state at either endpoint; the server returns to `LISTEN`, and the client closes the connection. If the server returned no TW-Negotiate option, the connection will use current TCP semantics: the side that issues the active close will enter `TIME-WAIT` (or both will if they close simultaneously).

This algorithm handles any non-simultaneous connection establishment; the following handles the simultaneous case. During a simultaneous open, neither endpoint is in the server role, so neither has the TW-Negotiate value has priority. As establishment progresses, both sides will find themselves in `SYN-RCVD` (the state transitions are `CLOSED`→`SYN-SENT`→`SYN-RCVD`) Each will know two TW-Negotiate values: theirs and the other endpoint's[1]. From here, each endpoint behaves as if it were a client in step 3 of the negotiation and had received the value in Table 1 from its peer. At most one endpoint will disagree with the conclusion, and send an `<RST>`.

This algorithm guarantees that the endpoints will agree on which will enter `TIME-WAIT` when the connection is dissolved, or will fall back to TCP semantics if either side does not support negotiation.

TW-Negotiation Values Known	TW-Negotiation Value To Use
Either Contains No Option	No Option
The Same IP Address	That IP Address
Two Different IP addresses	Larger IP Address

Table 1: Negotiation Values for Simultaneous Open

As an example of a negotiation when two endpoints simultaneously open the same connection, consider two endpoints, A and B. A's IP address is larger. Both always attempt to negotiate holding their own TIME−WAIT TCB, i.e., they send their own address in the TW-Negotiate option. The two endpoints attempt to open the connection, the <SYN>s cross in the network and both receive a <SYN> before they have received a <SYN, ACK>. The endpoints know the value of both TW-Negotiate options, but neither is in the server role above and can make the "final offer". they both act as if they had were clients and had sent a <SYN> with their preferred value, and received a <SYN, ACK> with A's address. They use A's address based on Table 1.

If having A hold the TIME−WAIT TCB is acceptable to both, they will both send an <SYN, ACK> with A's address in the header, and the connection will be established (after the final <ACK> exchange). If B is unwilling to have A hold the TCB, it will send an <RST> and the connection will never be established. A will always send a <SYN, ACK> because Table 1 has selected its preference; this will always be true of one endpoint, so only one will send the <RST>.

This system is biased toward endpoints with larger IP addresses; however, simultaneous attempts to establish connections are rare and never occur in client/server systems. Should systems evolve that exhibit frequent simultaneous connection establishment attempts and TIME−WAIT loading, the protocol can be modified to include a random number in each header and use that to pick the TCB holder.

When the a connection connection that has a negotiated TIME−WAIT holder is closed, the two endpoints will exchange <FIN> packets as usual, and the TIME−WAIT holder will enter TIME−WAIT and the other endpoint will enter CLOSED, regardless of which end closed actively and which (if either) passively.

When negotiation is added to a endpoint's operating system, most applications will use system-wide defaults for the TW-Negotiate option. These defaults will be set to minimize server load, i.e, to hold TIME−WAIT TCBs at clients. A mechanism, such as a socket option, will be provided to allow applications to override the default setting.

The negotiation algorithm allows busy servers to accept connections only from clients that are willing to incur the TIME−WAIT overhead. Application algorithms do not have to alter their use of connection close in their protocol, and incur no hidden performance penalty associated with the distribution of TIME−WAIT TCBs.

3.1. Barriers to Adoption

Although the algorithm above is simple to describe, it represents a significant change to the TCP state machine. Many TCP implementations are descended from the original BSD reference implementation of the TCP/IP stack that directly implements the TCP state machine[9]. Implementing changes to that state machine would require significant programming and testing effort. Furthermore, proofs of TCP correctness that rely on the TCP state machine would be invalidated.

The state machine changes reflect the fact that information from connection establishment affects the closure. Currently, once an endpoint has finished the three-way handshake and entered the ESTABLISHED state, it is impossible to tell what role it played in creating the connection. By using information from the connection establishment to determine the endpoints' behavior when the connection is terminated, we have created two states that represent an established connection, and which state a connection enters depends on the result of an option negotiation.

Negotiating the TIME−WAIT TCB holder when the connection is closed disrupts the state machine less, but reduces a endpoints' control over their resources. A client in a system that negotiates the holder before the connection is established cannot get the data it wants without reaching an agreement with the server about which side bears the costs of the TIME−WAIT TCB; a client in a system that negotiates the holder when the connection closes can always leave the server paying the TIME−WAIT cost. We prefer a system that makes agreement on the allocation of connection costs a prerequisite to incurring them. However, because allocating the TIME−WAIT state at connection close time is simpler, we have implemented an example of that system, which we discuss in Section 5.1.

We feel that the benefits of providing applications more control over the endpoint resources that they commit to a connection has significant advantages. The proposed system isolates that protocol behavior, which makes the solution more general than, for example, reversing the roles of active and passive closer. Finally it isolates application programs from an implementation detail of the transport protocol, allowing new application protocols to meet the needs of applications rather than being bent out of shape by transport.

4. Other Systems to Avoid TIME−WAIT TCB Loading

In this section we propose two less ambitious solutions to the server TIME−WAIT loading problem. Each solution is a small change to an existing protocol that reduces TIME−WAIT loading in HTTP and FTP servers. One system modifies TCP to exchange TIME−WAIT TCBs after a successful close, the other modifies HTTP to encourage clients to close the underlying TCP connection. We chose to modify HTTP because it is a large component of Internet traffic.

These solutions are intended to be practical ones. As such, they are incrementally deployable and compatible with existing protocol specifications. Both the TCP and HTTP solutions realize benefits without modifying the server systems, although additional benefits accrue if both client and server implement the HTTP changes. Neither set of changes violates the current TCP or HTTP specifications, so changed systems will operate in today's Internet.

We have implemented both systems, and observed that both significantly reduce the TIME-WAIT loading on HTTP servers. We discuss the performance of both systems in Section 5. Patches which implement the systems are available from the authors.

4.1. Transport Level (TCP) Solution

The TCP solution exchanges the TIME-WAIT state between the server and client when the connection is closed. We modify the client's TCP implementation so that after a successful passive close, it sends an <RST> packet to the server and puts itself into a TIME-WAIT state. The <RST> packet removes the TCB in TIME-WAIT state from the server; the explicit transition to a TIME-WAIT state in the client preserves correct TCP behavior.

If the client <RST> is lost, both server and client remain in TIME-WAIT state, which is equivalent to a simultaneous close. If either endpoint reboots during the <RST> exchange, the behavior is the same as if an endpoint running unmodified TCP fails with connections in TIME-WAIT state: packets will not be erroneously accepted if the endpoint recovers and refuses connections until a $2 \times$ MSL period has elapsed[1,7]. The behavior of an active close is unaffected.

Using an <RST> packet means that this system only works with TCP stacks that accept <RST> packets that arrive for a connection in TIME-WAIT state. Such stacks are susceptible to TIME-WAIT assassination[15], which can lead to connections becoming desynchronized or destroyed. TIME-WAIT assassination is the accidental or malicious deletion of a TIME-WAIT TCB at an endpoint, which can lead to confusion as shown in Figure 1.

Our system assassinates TIME-WAIT states at the server and replaces them at the client, which does not change TCP's behavior. Adding our system to a server that is susceptible to TIME-WAIT assassination does not make it more vulnerable, but a server that implements the changes in[15] to prevent assassinations will not benefit the system described in this section. Interactions between a server that prevents TIME-WAIT assassination and a client that implement our changes do not compromise TIME-WAIT guarantees.

Our system modifies the TCP state machine by changing the arc from LAST-ACK to CLOSED to an arc from LAST-ACK to TIME-WAIT and sending an <RST> when the arc is traversed. (See Figure 2 for the relevant section of the TCP state diagram.) To reduce TIME-WAIT loading from FTP or HTTP, these modifications need to be made only to clients.

Hosts that act primarily as clients may be configured with the new behavior for all connections; clients that serve as both client and server, such as HTTP proxies, may be configured to support both the new and old behaviors. Supporting both swapping and non-swapping close is straightforward, although it requires a more extensive modification of the TCP state machine.

To allow both behaviors on the same host we split the LAST-ACK state into two states, one that represents the current behavior (LAST-ACK) and one which represents the modified behavior (LAST-ACK-SWAP).[3] If the client invokes close while in CLOSE-WAIT, current TCP semantics apply; if the client invokes close_swap in the same state, the <RST>-sending behavior applies. Close and close_swap are indistinguishable if invoked from ESTABLISHED.

The state machine in Figure 3 implements both behaviors. Compare it with the earlier Figure 2, which shows the state machine for TCP's connection closing behavior. The details of active and simultaneous closing are unchanged from Figure 2, and are omitted for clarity.

Adding close_swap does not require adding a system call. One implementation of close_swap adds a per-connection flag that changes the default behavior when set. When a connection with the flag set is closed, the close system call calls close_swap instead. Endpoints that are primarily clients set their default close behavior to be close_swap, endpoints that are primarily servers will default to close.

The performance of this system is limited by how efficiently the endpoint processes <RST> packets. Endpoints that incur a high overhead to handling <RST>s, or delay processing them are not good candidates for this approach.

This system also changes the meaning of the <RST> packet. An <RST> packet currently indicates an unusual condition or error in the connection; this system proposes making it part of standard connection closing procedure.

The TCP solution described in Section 3 does not suffer from the drawbacks associated with using <RST> packets, either in terms of exposing systems to incorrect TCP semantics or in terms of additional processing time for <RST> packets.

---

[3] These states may both be reported as LAST-ACK to monitoring tools for backward compatibility.

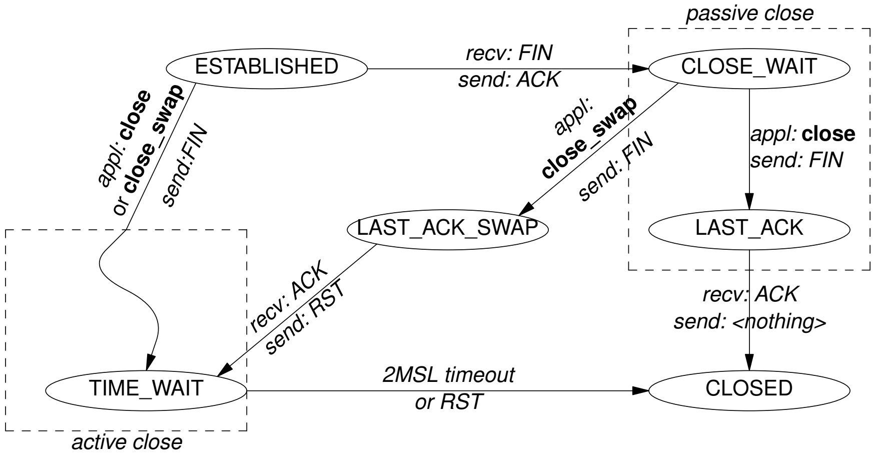

Figure 3: Modified TCP State machine for swapping `TIME-WAIT` states

4.2. Application Level Solution for HTTP

The systems in Section 3 and Section 4.1 both involve changes to the transport protocol which is used by many applications. Directly modifying an application protocol that is loading servers may control the loading problem and minimize the effect on other applications.

This section describes modifications to HTTP that alleviate the contribution of that protocol to `TIME-WAIT` loading. We chose to modify HTTP because it is a major source of client/server Internet traffic.

Early versions of HTTP relied on the closing of the TCP connection to indicate the end of a transaction. Among the changes is HTTP 1.1[2] is the support of persistent connections, a technique that allows clients to pass multiple transactions over the same TCP connection. In order to support persistent connections, the end-of-connection and end-of-transaction indications have been decoupled. This decoupling allows us to modify HTTP to allow clients to actively close connections and therefore hold the `TIME-WAIT` state.

We modify HTTP 1.1 to include a notification from the client that the connection is closed. This notification takes the form of an extension request, called `CLIENT_CLOSE`. An extension request is a new HTTP command, like `PUT` or `POST`, that is not explicitly defined in the HTTP specification[2] A `CLIENT_CLOSE` request requires no reply. It terminates a series of requests on a persistent connection, and indicates to the server that the client has closed the TCP connection. A client will close the TCP connection immediately after sending the `CLIENT_CLOSE` request to the server.

A `CLIENT_CLOSE` request differs from including a `Connection: close` in the header of a request because a request that includes `Connection: close` still requires a reply from the server, and the server will (actively) close the connection[2]. A `CLIENT_CLOSE` request indicates that the client has severed the TCP connection, and that the server should close its end without replying.

`CLIENT_CLOSE` is a minor extension to the HTTP protocol. Current HTTP clients conduct an HTTP transaction by opening the TCP connection, making a series of requests with a `Connection: close` line in the final request header, and collecting the responses. The server closes the connection after sending the final byte of the final request. Modified clients open a connection to the server, make a series of requests, collect the responses, and send a `CLIENT_CLOSE` request to the server after the end of the last response. The client closes the connection immediately after sending the `CLIENT_CLOSE`.

Modified clients are compatible with the HTTP 1.1 specification[2]. A server that does not understand `CLIENT_CLOSE` will see a conventional HTTP exchange, followed by a request that it does not implement, and a closed connection when it tries to send the required error response. A conformant server must be able to handle the client closing the TCP connection at any point. The client has gotten its data, closed the connection and holds the `TIME-WAIT` TCB.

We intend to extend `CLIENT_CLOSE` to include a mechanism for the server to request that the client close the connection. This is analogous to the current `Connection: close` but is initiated by the server and implemented by the client. Under HTTP 1.1, loaded servers are allowed to close persistent connections to reduce their load, but they will incur a `TIME-WAIT` TCB by doing so. Allowing servers to request that the client disconnect sheds the `TIME-WAIT` load at the server as well.

Modifying servers to recognize `CLIENT_CLOSE` can make parts of their implementation easier. Mogul et al. note that discriminating between a persistent connection that is temporarily idle and one that is closed can be difficult for servers because many operating systems do not notify the server that the client has closed the connection until the server tries to read from it[2]. `CLIENT_CLOSE` marks closing connections, which simplifies the server code that detects and closes connections that clients have closed.

Having a client decide when to initiate a CLIENT_CLOSE is somewhat complex. It has to consider user browsing patterns, state of local resources, and the state of server resources. The last may be the trickiest to incorporate. As mentioned above, servers may choose to terminate persistent connections in order to reuse the resources allocated to that connection. Servers need a mechanism to communicate their commitment level to a connection, so that clients and servers are more likely to decide to terminate the same ones[16].

The CLIENT_CLOSE request has been implemented directly in the apache-1.2.4 server[17] and test programs from the WebSTONE performance suite[18]. Patches are available from the authors.

Adopting the HTTP solution is effective if HTTP connections are the a major source of TIME-WAIT loading; however, if another protocol begins loading servers with TIME-WAIT states, that protocol will have to be modified as well. Currently, we believe HTTP causes the bulk of TIME-WAIT loading.

The CLIENT_CLOSE system requires changes only on the client side, although making servers aware of CLIENT_CLOSE may enhance the system's effectiveness. The system conforms to the HTTP 1.1 specification and requires no changes to other protocols. to our knowledge, it creates no new security vulnerabilities.

5. Experiments

In this section we present experiments that demonstrate TIME-WAIT loading and show that our solutions reduce its effects. The proposed solutions have been implemented under SunOS 4.1.3 and initial evaluations of their performance have been made using both custom benchmark programs and the WebSTONE benchmark[18]. The tests were run on workstations connected to the 640 Mb/sec Myrinet LAN.

We performed two experiments. The first experiment shows that TCB load degrades server performance and that our modifications reduce that degradation. The second illustrates that both our TCP and HTTP solutions improve server performance under the WebSTONE benchmark, which simulates typical HTTP traffic. The last experiment shows that our modifications enable a server to support HTTP loads that it cannot in their default configurations.

5.1. Demonstration of Worst-Case Server Loading

The first experiment was designed to determine if TCB load reduces server throughput and if our modifications alleviate that effect. This experiment used four Sparc 20/71's connected by a Myrinet using a user-level data transfer program over TCP. The throughput is the average of each of two client workstations doing a simultaneous bulk transfer to the server. We varied the number of TIME-WAIT TCBs at the server by adding TIME-WAIT states.

The procedure was:

1. Two client machines establish connections to the server
2. The server is loaded with TIME-WAIT TCBs state by a fourth workstation. This workstation established and shut down connections as fast as possible until the server was loaded.
3. The two bulk transport connections transfer data. (Throughput timing begins when the data transfer begins, not when the connection is established. TIME-WAIT TCBs may expire during the transfer.)
4. Between runs, the server was idled until all TIME-WAIT TCBs timed out.

The results are plotted in Figure 4. Each point is the average of ten runs, error bars are standard deviations. The "Modified TCP" curve represents a SunOS system with the TCP modifications from Section 4.1. "Unmodified TCP" represents an unmodified SunOS system. All other parameters remained unchanged.

The experimental procedure is designed to isolate a worst case at the server. The client connections are established first to put them at the end of the list of TCBs in the server kernel, which will maximize the time needed to find them using SunOS's linear search. Two clients are used to neutralize the simple caching behavior in the SunOS kernel, which consists of keeping a single pointer to the most recently accessed TCB. Two distinct clients are used to allow for bursts from the two clients to interleave; two client programs on the same endpoint send bursts in lock-step, which reduces the cost of the TCB list scans.

The experiment shows that under worst case conditions, TCB load can reduce throughput by as much as 50%, and that

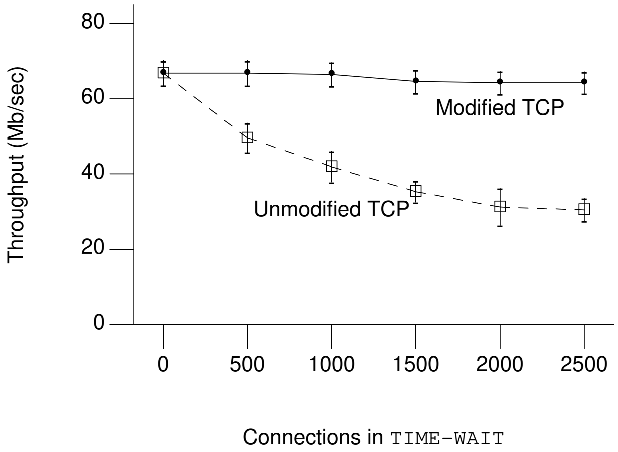

Figure 4: Worst-Case Server Loading

our TCP modifications improve performance under those conditions.

While it is useful that our modifications perform well in the worst case, it is important to asses the worth of the modifications under expected conditions. The previous experiment constructed a worst case scenario; the following experiment uses WebSTONE to test our modifications under more typical HTTP load.

5.2. HTTP Load Experiments

WebSTONE is a standard benchmark used to measure web server performance in terms of connection rate and per connection throughput. To measure server performance, several workstations make HTTP requests of a server and monitor the response time and throughput. A central process collects and combines the information from the individual web clients. We modified the benchmark to measure the amount of memory consumed by TCBs on the server machine. We used WebSTONE version 2 for these experiments. The same workstations and network from Section 5.1 are used in these experiments.

WebSTONE models a heavy load that simulates HTTP traffic. Two workstations run multiple web clients which continuously request files ranging from 9KB to 5MB from the server. Each workstation runs 20 web clients. TCP modifications are as described in Section 4.1 and HTTP modifications are as described in Section 4.2. Results are shown in Table 2.

Both modifications show marked improvements in throughput, connection rate and memory use. TCP modifications increase connection rate by 25% and HTTP modifications increase connection rate by 50%. Server memory requirements are reduced regardless of the HTTP/TCP demultiplexing implementation.

When clients request smaller files, unmodified systems fail completely because they run out of memory; systems using our modifications can support much higher connection rates than unmodified systems. Table 3 reports data from a typical WebSTONE run using 8 clients on 4 workstations connecting to a dedicated server. All clients request only 500 byte files.

System Type	Throughput (Mb/sec)	Conn. per second	TCB Memory (Kbytes)
Unmodified	20.97	49.09	722.7
TCP Mods.	26.40	62.02	23.1
HTTP Mods.	31.73	74.70	23.4

Table 2: `TIME-WAIT` Loading Under WebSTONE

System Type	Throughput (Mb/sec)	Conn. per second	TCB Memory (Kbytes)
Unmodified	fails	fails	fails
TCP Mods.	1.14	223.8	16.1
HTTP Mods.	1.14	222.4	16.1

Table 3: `TIME-WAIT` Loading Under WebSTONE With Small Files

The experiments support the hypothesis that the proposed solutions reduce the memory load on servers. The worst-case experiment shows that the system with a modified TCP performs much better in the worst case, and that server bandwidth loss can be considerable. The WebSTONE experiments shows that both systems reduce memory usage, and that this leads to performance gains. Finally modified systems are able to handle workloads that unmodified systems cannot.

This is a challenging test environment because the TCB load of the server workstations is spread across only two clients rather than the hundreds that would share the load in a real system. The clients suffer some performance degradation due to the accumulating TCBs, much as the server does in the unmodified system.

5.3. `TIME-WAIT` Avoidance and Persistent Connections

The systems proposed are compatible with the per-connection schemes such as persistent connections. To show that our systems improve the memory performance of those systems, we ran WebSTONE experiments using persistent connections and our systems. The experiment used the same network as the experiments described in Section 5.2; two workstations acted as clients, and one as a web server. Each client used the same request pattern as the results in Table 2. Each client issued 5 HTTP requests, waited until they all arrived, and sent 5 more. Each connection served 10 requests in two

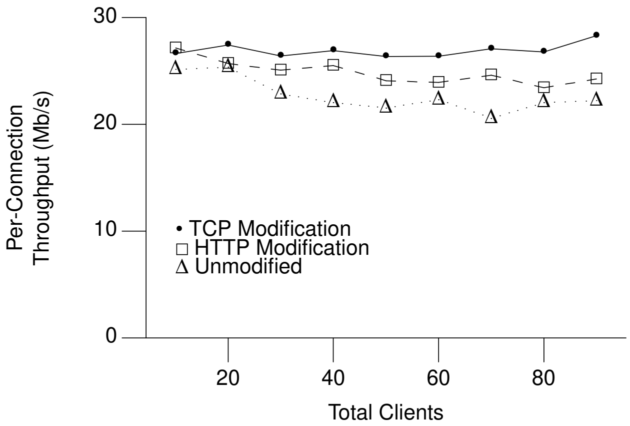

Figure 5: Throughput vs. Clients (Persistent Connections)

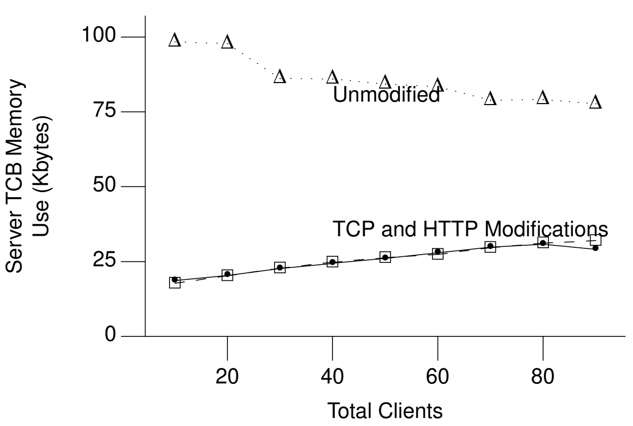

Figure 6: Memory Use vs. Clients (Persistent Connections)

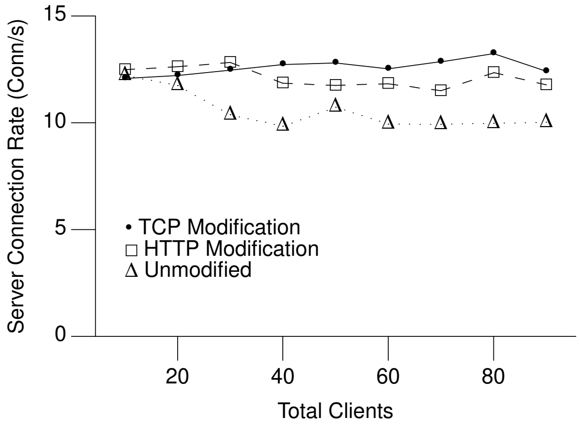

Figure 7: Connection Rate vs. Clients (Persistent Connections)

5-request bursts.

Figure 5 shows how per-connection average client throughput varies with increasing number of clients. Connection throughputs are comparable to those in Table 2, with the difference due primarily to the longer life of these connections. For example, congestion windows will open farther. Our TIME−WAIT avoidance methods increase the per-connection throughput as client load increases.

Figure 6 shows that in addition to a modest increase in per-connection throughput, our systems provide significant reduction in the memory used for TCB blocks at servers. That figure plots the number of TCBs in use by the server versus the client load.

It appears from Figure 6 that a simple persistent connection system is showing improved memory performance with increasing client load, but this is not the case. Figure 7 shows that the connection rate decreases with increasing client load in the unmodified system, due to the additional overhead. The memory usage follows the connection rate in the unmodified system. Because Figure 6 includes active TCBs as well as TIME−WAIT TCBs, our systems show a linear increase in used TCBs.

6. Conclusions

We have described how TCP interacts with certain application protocols to load servers with TIME−WAIT TCBs, and shown examples of this behavior. This interaction is a direct result of the the simplifying assumption that an endpoint's role in closing the connection is independent of its role in establishing the connection.

We have proposed negotiating which endpoint holds the TIME−WAIT state during connection establishment, and proposed a negotiation protocol. This system extends TCP functionality to allocate TIME−WAIT TCBs to the proper end of the connection without interfering with the semantics of the application protocols or leaving known vulnerabilities in TCP. However, because our proposal involves significant changes to the TCP stack we expect some resistance to its adoption.

We have also implemented and tested a simpler TCP solution and an HTTP solution to show shift the TIME−WAIT load from client to server. We have presented experimental evidence that TIME−WAIT loading can affect server performance under SunOS 4.1.3. Under these conditions, throughput can be reduced by as much as 50%.

Using WebSTONE, we have shown that HTTP clients and servers using one of our systems exhibit higher throughputs under SunOS 4.1.3. Clients and servers using our system can support higher connection rates than unmodified systems, in certain configurations.

We have shown that our systems combined with persistent HTTP connections use less memory than an otherwise unmodified SunOS 4.1.3 system using persistent connections for a given client load. Our systems interoperate with persistent connections.

Although there are other systems that address the throughput problems we discuss here[11,12], our systems attack the memory loading problem directly. This can reduce the cost of deploying a server, which can be especially important to an embedded or battery powered server.

Table 4 compares the three systems proposed here.

We believe that TCP should eventually be modified to support TIME−WAIT negotiation. The coming upgrade from IP version 4 to version 6 represents an opportunity to revisit TCP implementation and design as well. This would be an opportune time to include TIME−WAIT state negotiation.

We have also proposed two effective, practical systems for reducing TW load at servers until the more ambitious proposal can be implemented. Adopting one of them would also reduce TIME−WAIT loading at servers.

	TCP With TIME−WAIT Negotiation	TCP With Client <RST>	CLIENT_CLOSE HTTP Extension
Reduces TIME−WAIT Loading	Yes	Yes	Yes
Compatible With Current Protocols	Yes	Yes	Yes
Changes Are Effective If Only The Client Is Modified	No	Yes	Yes
Allows System To Prevent TIME−WAIT Assassination	Yes	No	Yes
No Changes To Transport Protocol	No	No	Yes
No Changes To Application Protocols	Yes	Yes	No
Adds No Packet Exchanges To Modified Protocol	Yes	No	No
TIME−WAIT Allocation Is A Requirement of Connection Establishment	Yes	No	No

Table 4: Summary of Proposed Systems

References

1. Jon Postel, ed., "Transmission Control Protocol," *RFC-793/STD-7* (September, 1981).
2. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext Transport Protocol - HTTP/1.1," *RFC-2068* (January, 1997).
3. Robert G. Moskowitz, "Why in the World Is the Web So Slow," *Network Computing,* pp. 22-24 (March 15, 1996).
4. J. Postel and J. K. Reynolds, "File Transfer Protocol," *RFC-959,* USC/Information Sciences Institute (October, 1985).
5. Sun Microsystems, Inc., "Remote Procedure Call Specification," *RFC-1057* (June 1, 1988).
6. Jon Postel, ed., "Internet Protocol," *RFC-791/STD-5* (September 1981).
7. Internet Engineering Task Force, R. Braden, ed., "Requirements for Internet Hosts – Communications Layers," *RFC-1122* (October 1989).
8. W. Richard Stevens, *TCP/IP Illustrated, Volume 1, The Protocols,* Addison-Wesley, Reading, MA, et al. (1994).
9. Gary R. Wright and W. Richard Stevens, *TCP/IP Illustrated, Volume 2 The Implementation,* Addison-Wesley, Reading, MA, et al. (1995).
10. Myricom, Inc., Nannette J. Boden, Danny Cohen, Robert E. Felderman, Alan E Kulawik, Charles L. Seitz, Jakov N. Selovic, and Wen-King Su, "Myrinet: A Gigabit-per-second Local Area Network," *IEEE Micro,* pp. 29-36, IEEE (February 1995).
11. Mike Karels and David Borman, *Personal Communication* (July 1997).
12. Paul E. McKenney and Ken F. Dove, "Efficient Demultiplexing of Incoming TCP Packets," *Proceedings of SIGCOMM 1992,* vol. 22, no. 4, pp. 269-279, Baltimore, MD (August 17-20, 1992).
13. Hendrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley, "Network Performance Effects of HTTP/1.1, CSS1, and PNG," *Proceedings of the SIGCOMM Symposium on Applications, Technologies, Architectures, and Protocols for Computer Communication,* pp. 155-166, Cannes, France (14-18 September 1997).
14. Van Jacobson, Robert Braden, and D. Borman, "TCP Extensions for High Performance," *RFC-1323* (May 1992).
15. R. Braden, "TIME−WAIT Assassination Hazards in TCP," *RFC-1337,* USC/Information Sciences Institute (May 1992).
16. James Gettys, *Personal Communication* (December 1997).
17. Roy T. Fielding and Gail Kaiser, "Collaborative Work: The Apache Server Project," *IEEE Internet Computing,* vol. 1, no. 4, pp. 88-90, IEEE (July/August 1997).
18. Gene Trent and Mark Sake, "WebSTONE: The First Generation in HTTP Server Benchmarking," *white paper,* Silicon Graphics International (February 1995).