INTERNET-DRAFT
Expires: February 10, 1998
draft-faber-time-wait-avoidance-00.txt

T. Faber
J. Touch
W. Yue
USC/ISI
August 1997

Avoiding the TCP TIME_WAIT state at Busy Servers

Status of this Memo

Abstract

This document describes the problems associated with the accumulation of TCP TIME_WAIT states at a network server, such as a web server, and details two methods for avoiding that accumulation. Servers that have many TCP connections in TIME_WAIT state experience performance degradation, and can collapse. One solution is a TCP modification that causes clients to enter TIME_WAIT state rather than servers. The other is an HTTP modification that allows the client to close the transport connection, maintaining the TIME_WAIT state at the client. The goal of both approaches is ensure that TIME_WAIT states accumulate at the less loaded endpoint.

The document also presents initial performance data from reference implementations of these solutions, which suggest that the modifications improve HTTP connection rates at the server by as much as 50%, and allow servers to operate at small transaction throughputs that they cannot sustain their default configuration.

Introduction

This draft describes the causes and effects of TIME_WAIT TCP protocol control block (TCB) accumulation at servers and proposes independent application and transport level modifications that remove that buildup. We present experimental results showing a 50% improvement in HTTP connection rates, as measured by WebSTONE[1], as well as evidence that modified servers function at higher loads than unmodified servers can.

TIME_WAIT state and its effects

TCP includes a mechanism to ensure that packets associated with one connection that are delayed in the network are not accepted by later connections between the same hosts[2]. The mechanism is implemented by the TIME_WAIT state of the TCP protocol. When an endpoint closes a TCP connection, it keeps state about that connection, usually a copy of the TCB, for twice the maximum segment lifetime (MSL). A connection in this state is in TIME_WAIT, and the endpoint holding the TIME_WAIT TCB rejects any packets addressed to the TIME_WAIT connection from the other endpoint.

Keeping this TIME_WAIT TCB at either of the hosts prevents a new connection with the same combination of source address, source port, destination address, destination port from being created. Either endpoint being in TIME_WAIT prevents data transfer on the connection, so protocol correctness is unaffected by which host holds the TIME_WAIT TCB. Our modifications center on ensuring that the TIME_WAIT TCB is on the less loaded endpoint.

Heavily loaded servers potentially keep thousands of TIME_WAIT TCBs, which 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. There are a finite number of mbufs available in the system, and mbufs consumed by TCBs cannot be used for other purposes, e.g., to move data. Certain systems on high speed networks 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/sec Myrinet[3] cannot support more than 60 connections/sec.

Incoming packets must be demultiplexed by finding the receiving connection in the host's TCB list. This process can be slowed when the TCB list is full of TIME_WAIT TCBs. In the simplest implementation, the TCB list is searched linearly to demultiplex the incoming packet to the appropriate connection, which can make TCB lookup a bottle-neck. The additional search overhead can cut throughput between two SunOS 4.1.3 SPARCStations on a Myrinet in half.

Other Proposed Solutions

There are other solutions to the increased lookup overhead problem, e.g., storing all TIME_WAIT TCBs at the end of the list and using them as a search terminator as BSDI's BSD/OS does[4], or hashing TCBs rather than keeping them in a list[5]. These solutions do not address the loss of memory due to accumulation of TIME_WAIT states, so servers may still be unable to serve a high client load. These approaches improve system response until the server collapses due to lack of free mbufs; our approach of removing the TIME_WAIT state from the server eliminates this cause of server collapse.

Allocating more memory to system mbufs or reducing the amount of data cached per connection allows servers to function under a higher load before collapsing. The servers' performance will continue to degrade. Moving TIME_WAIT to clients removes this cause of system degradation and collapse without changing resource allocations.

The costs of accumulating TIME_WAIT TCBs have become more apparent as HTTP becomes more prevalent. Under HTTP 1.1, servers terminate con-nections by closing the underlying TCP connection[6], which results in accumulation of TCBs at servers[7].

HTTP 1.1 reduces the number of connections per transaction using per-sistent connections; however, with respect to TIME_WAIT buildup, the use of persistent connections[6] is similar to adding more memory to servers: servers can support a larger load before the effect becomes noticeable, but performance eventually degrades. Servers supporting persistent connections can support more transactions per connection, and will benefit from our modifications by being able to support more connections.

Our Proposed Solutions

Because the accumulation of TIME_WAIT TCBs is caused by the interac-tion between transport and application protocols, modifications can be made to either protocol to to alleviate it. Changing the transport protocol confers the benefits to more applications, but there may be more resistance to changing a protocol on which many applications depend. Application level changes restrict the benefits (and drawbacks) to the application for which the solution is implemented. Furthermore, application solutions are not always possible; for example, protocols that use the closing of a transport connection to indicate end-of-file are not good candidates for removing TIME_WAIT TCBs at the application layer.

This document proposes distinct extensions to TCP and to HTTP that allow hosts to control which end of the connection remains in

TIME_WAIT state. A solution needs to be implemented at only one level, transport or application. We describe and measure both to have a basis for comparison. Preliminary experiments indicate that both systems reduce the memory usage of web servers due to TIME_WAIT states to negligible levels, with accompanying performance improvements. The TCP modifications require only client side changes, and can be deployed incrementally. The HTTP changes affect client and server, but are compatible with HTTP 1.1 behavior, and can also be incrementally deployed.

The remainder of this document is organized as follows: it presents the two proposed solutions, compares them, discusses the results of initial experiments with the solutions, and draws conclusions and outlines future work.

Transport Level (TCP) Solution

The TCP solution exchanges the TIME_WAIT state between the server and client. We modify the client's TCP implementation so that after it has completed a passive close of a transport connection, 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 also ensures correct behavior and is equivalent to a simultaneous close in the current protocol. If either host reboots during the RST exchange, the behavior is the same as if a host running unmodified TCP fails with connections in TIME_WAIT state: packets will not be erroneously accepted if the host recovers and refuses connections until a 2 MSL period has elapsed[2].

More formally, the change to the TCP state machine results in 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. 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, for example proxies, may be configured to support both behaviors. The implementation of both behaviors is straightforward, although it requires a more extensive modification of the TCP state machine.

Allowing both behaviors on the same host requires splitting 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). These states may both be reported as LAST_ACK to monitoring tools. The state machine determines which state to enter

from CLOSE_WAIT based on whether the application issues a close or a close_swap.

The current passive close path is:

```
server                                client
-----------------------------------------------------------
ESTABLISHED                           ESTABLISHED
(get application close)
goto FIN_WAIT_1
send FIN           ---FIN--->
                                      goto CLOSE_WAIT
                  <---ACK---          send ACK
goto FIN_WAIT_2
                                      (get application close)
                                      goto LAST_ACK

                  <---FIN---          send FIN
goto TIME_WAIT
send ACK           ---ACK--->
                                      goto CLOSED
```

This solution adds this branch from CLOSE_WAIT on the client side:

```
   server                            client
   ------------------------------------------------------------
   ESTABLISHED                       ESTABLISHED
   (get application close)
   goto FIN_WAIT_1
   send FIN           ---FIN--->
                                     goto CLOSE_WAIT
                     <---ACK---      send ACK
   goto FIN_WAIT_2
                                     (get application close_swap)
                                     goto LAST_ACK_SWAP
                     <---FIN---      send FIN
   goto TIME_WAIT
   send ACK           ---ACK--->
                                     goto TIME_WAIT
                     <---RST---      send RST
   goto CLOSED
```

Strictly speaking, the transition of the client to TIME_WAIT is extraneous, because any host sending an RST is obligated not to allow a connection between the same pair of addresses and ports for a time of at least 2 MSL.

Distinguishing between close and close_swap does not require changing the application interface. For example, a per-connection flag can be added to change the default behavior, where the default behavior is chosen based on whether the host is primarily a client or a server. Hosts that are primarily clients will follow the close_swap path unless overridden and servers the close path.

Implementations of this system do not change the API at all if all connections from the same host have the same semantics; hosts which are primarily clients will see no change. Only hosts that support both semantics will have a change to the API, and this will be an additional socket option or similar small change.

The solution we propose is designed to interoperate with the existing TCP specification. A cleaner implementation of our solution would be to change both endpoint implementations to negotiate which endpoint maintains the TIME_WAIT TCB. However this would require changing all TCP implementations, which ours does not.

A SunOS 4.1.3 patch is available from the authors.

Application Level Solution for HTTP

Protocols that use the state of the transport connection as signalling dictate which endpoint closes a connection, and therefore which incurs the cost of the TIME_WAIT TCB. For example, early HTTP servers used the state of the transport connection as an implicit indicator of both transaction lifetime and request length. The server closing the TCP connection indicated to the client that the whole response had arrived, and, because there were no persistent connections, that the HTTP exchange was over. Because the server was using the close to mark the end of both the transaction and the exchange, it was required to initiate the close.

HTTP 1.1 has sufficient framing to allow a modification to shifting TIME_WAIT TCBs to the clients[6]. Responses are self-delineating; all responses include the size of the response either in the headers or via the chunking mechanism. When using persistent connections, which is the default behavior in HTTP/1.1, requests have fields which can be used to control the transport connection. The server is no longer required by the protocol to close the transport connection.

To control the distribution of TIME_WAIT TCBs from the application level, our HTTP modifications arrange that the client closes the TCP connection. This requires the client to be able to detect the end of a response. Under HTTP 1.1, this information is available to the client as a side effect of persistent connections. We advocate a change in client behavior which requires them to close the transport connection underlying an HTTP connection, and an extension of the request format which allows the client to notify the server that it is breaking the TCP connection.

We propose adding a CLIENT_CLOSE request to HTTP that indicates that a client is ending the HTTP exchange by closing the underlying TCP connection. 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 initiate an active close on 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 (passively) close the connection[6]. A CLIENT_CLOSE request indicates that the client has severed the TCP connection, and that the server should close its end without replying.

Incorporating the CLIENT_CLOSE into the transaction 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[6]. 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 an 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.

Modifying servers to recognize CLIENT_CLOSE can make parts of their implementation easier. Mogul et al. note that detecting closed connections can be difficult for servers[6]. CLIENT_CLOSE marks closing connections, which simplifies the server code that detects and closes

connections that clients have intentionally closed.

The CLIENT_CLOSE request has been implemented directly in the apache-1.24[8] server and test programs from the WebSTONE performance suite. Patches are available from the authors.

Initial Implementation

In this section we present experiments that demonstrate the problem and show our solutions effectiveness. 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. The tests were run on hosts 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.

The first experiment was designed to determine if TCB load reduces server performance and if our modifications alleviate it. This experiment used four Sparc 20/71's across the Myrinet using a user-level data transfer program over TCP. The throughput is the average of each of two client hosts doing a simultaneous bulk transfer to the server host. We vary the number of TIME_WAIT TCBs at the server by adding dummy TIME_WAIT states.

The experiment was:

1. Two client machines establish connections to the server

2. The server is loaded with extraneous TIME-WAIT TCBs state by a fourth host.

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 is allowed to idle and remove TCBs, to control conditions for all runs.

Each result is the average of ten runs.

Connections in TIME_WAIT	Throughput (Mb/sec) (Unmodified)		Throughput (Mb/sec) (with TCP Modification)	
	avg.	std. dev.	avg.	std. dev.
0	66.8	3.3	66.8	3.3
500	49.6	3.9	66.8	3.3
1000	41.9	4.1	66.5	3.1
1500	35.3	2.8	64.6	3.0
2000	31.2	4.9	64.3	3.0
2500	30.5	3.0	64.3	2.9

Table 1: Worst case throughput experiment

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 host 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 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.

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. The benchmark has been augmented to measure the amount of memory consumed by TCBs on the server machine. We used WebSTONE version 2 for these experiments.

WebSTONE models a heavy load that simulates HTTP traffic. Two hosts run multiple web clients which continuously request files ranging from 9KB to 5MB from the server. Each host runs 20 web clients.

Results from a typical run using clients are summarized:

System Type	Throughput (Mb/sec)	Connections per second	TCB Memory Use (Kbytes)
Unmodified	20.97	49.09	722.7
TCP Modification	26.40	62.02	24.1
HTTP Modifications	31.73	74.70	24.4

Table 2: WebSTONE benchmark with large fileset

Both modifications show marked improvements in throughput and connection rate. TCP modifications increase connection rate by 25% and HTTP modifications increase connection rate by 50%. We believe the TCP modification is less effective than the HTTP modification because it adds another packet exchange. Packet traces are being used to confirm this. [note: more will be included on this in later drafts]

When more 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. The following table reports data from a typical WebSTONE run using 8 clients on 4 hosts connecting to a dedicated server. All clients request only 500 byte files.

System Type	Throughput (Mb/sec)	Connections per second	TCB Memory Use (Kbytes)
Unmodified	fails	fails	fails
TCP Modification	1.14	223.8	16.1
HTTP Modifications	1.14	222.4	16.1

Table 3: WebSTONE benchmark with small files

The experiments support the hypothesis that the proposed solutions reduce the memory load on servers. The custom benchmark shows that the system with a modified transport performs much better in the worst case, and that server bandwidth loss can be considerable. The WebSTONE benchmark 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 host is spread across only two client hosts 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.

Comparison of Methods

The primary contrast between the TCP solution and the HTTP solution is that they are implemented at different levels of the protocol hierarchy. The TCP solution has the benefits and drawbacks of a transport level solution: it applies the fix transparently to all application protocols running over TCP, but may be difficult to adopt for the same reason. A change to TCP affects many applications, and many resist changes to TCP to avoid unintended consequences. The HTTP solution has the trade-offs of an application modification: only HTTP will exhibit the new behavior, and the behavior of other applications will be limited. If another protocol causes a TIME_WAIT state buildup, an HTTP fix will not prevent it.

The performance of our TCP modification will also be limited by how efficiently hosts process RST packets. Hosts that incur a high overhead to handling RSTs, or delay processing them, will not perform as well. This may be one reason that the TCP solution shows less improvement than the HTTP solution in the small file experiment above. [note this will be expanded upon]

The meaning of the RST packet is also changed by out TCP solution. An RST packet is intended to indicate an unusual condition or error in the connection. We are proposing making it part of standard operating procedure. The change in semantics of the RST packet is a result of maintaining compatibility with current TCP. Some browsers are currently using RST in unintended ways as well.

Ideally, the two TCP endpoints would negotiate which would hold the TIME_WAIT TCB during connection establishment, but this would require changing the TCP packet format to allow room for that negotiation, and further changes to the state machine. We believe such a system is the best solution to the TIME_WAIT TCB accumulation problem, but recognize that such a large change to TCP would be difficult to get adopted.

Adopting the HTTP solution is effective if HTTP connections are the source of TIME_WAIT loading; however, if another protocol begins loading servers with TIME_WAIT states, that protocol will have to be fixed as well. Currently, we believe HTTP causes the bulk of TIME_WAIT loading, which is why we chose to implement our solution under HTTP; in the future other protocols may be the source.

Not adopting a TCP fix means that future protocols should be designed to control TIME_WAIT loading, which will constrain their semantics. Specifically, application protocols will not be able to use the state of the transport connection as implicit signalling; application layer protocols will be constrained to include framing and connection

control information or run the risk of TIME_WAIT loading servers. For example, streaming real-time transmission systems may make use of such implicit signalling.

Some existing protocols, such as FTP[9], make use of implicit signalling, and cannot be retrofitted with TIME_WAIT controls. As these protocols are currently used, they do not appear to be major sources of TIME_WAIT loading. They could become important to TIME_WAIT load if a protocol has a resurgence or is used in new ways, or if its smaller loading characteristics become significant after the HTTP load is reduced. If that happens, a backward-compatible solution may not be possible.

Both the TCP and the HTTP solutions are incrementally deployable and solve the problem at hand. Which to deploy in the Internet depends on how the community weighs the changing the semantics of the existing transport protocol versus restricting the semantics of future application protocols.

Conclusions

This document has discussed the problem of server memory load due to terminated connections remaining in TIME_WAIT state. Servers can become so memory poor at high connection rates that they are unable to transfer data at all. Even if servers can continue to function, their performance can suffer.

Two solutions to the memory load problem have been presented at the transport (TCP) level and the application (HTTP). Both solutions allow a client to take on its share of the server memory load. The transport level solution adds a new state and operation to the TCP state machine that explicitly moves the TIME_WAIT state from active close initiator to passive closer. The application level solution adds an access method to HTTP that allows a client to notify a server that it is actively closing the connection, and maintaining the TIME_WAIT state.

Both solutions will interoperate with existing systems, allowing for easy deployment. Patches are available from the authors for both solutions: TCP modifications are available for SunOS 4.1.3 and HTTP modifications are available for apache-1.24.

Although there are certainly other methods of dealing with TIME_WAIT state accumulation, the methods presented here have the benefits that they preserve current TCP behavior, are incrementally deployable, and are small simple changes to existing systems. Most other solutions, such as ending connections with an RST or moving TIME_WAIT TCBs to other internal queues at the server, either break transport behavior,

or do not address the memory load problem directly.

Security Considerations

The practices advocated in this paper do not seem to affect the security of either the HTTP or TCP protocols.

The increased use and change in semantics of RST packets may cause false alarms in systems that monitor them.

Authors' Addresses

Ted Faber
Joseph Touch
Wei Yue
University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695

USA
Phone: +1 310 822 1511
Fax: +1 310 823 6714
EMail: faber@isi.edu
       touch@isi.edu
       wyue@isi.edu

This draft expires March 20, 1998.

References

1. Gene Trent and Mark Sake, "WebSTONE: The First Generation in HTTP Server Benchmarking," white paper, Silicon Graphics International (February 1995), available electronically at <http://www.sgi.com/Products/WebFORCE/WebStone/paper.html>.

2. Jon Postel, "Transmission Control Protocol," RFC-793/STD-7 (September, 1981).

3. 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).

4. Mike Karels and David Borman, Personal Communication (July 1997).

5. 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).

6. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext Transport Protocol - HTTP/1.1," RFC-2068 (January, 1997).

7. Robert G. Moskowitz, "Why in the World Is the Web So Slow," Network Computing, pp. 22-24 (March 15, 1996).

8. 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), available electronically at <http://www.computer.org/internet/ic1997/pdf/w4088.pdf>.

9. J. Postel and J. K. Reynolds, "File Transfer Protocol," RFC-959
   (October, 1985).