Improving HTTP Performance Using “Stateless” TCP

David A. Hayes
Centre for Advanced Internet Architectures,
Swinburne University of Technology
Melbourne, Australia
david.hayes@ieee.org

Michael Welzl
Department of Informatics,
University of Oslo, Norway
michawe@ifi.uio.no

Grenville Armitage, Mattia Rossi
Centre for Advanced Internet Architectures,
Swinburne University of Technology
Melbourne, Australia
{garmitage,mrossi}@swin.edu.au

ABSTRACT

TCP is quite a heavyweight protocol when serving very small web pages. We introduce a server-side kernel modification which enables a web server to perform HTTP over a UDP socket while the kernel provides a regular TCP interface ‘on the wire’ to remote clients. We show that our “stateless” TCP modification can greatly reduce a server’s CPU usage (> 20,%) and TCP related memory requirements($> 90\,\%$), potentially enabling it to serve small web pages even under extreme overload conditions.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Distributed Systems—*Network operating systems*

General Terms

Performance

Keywords

HTTP, TCP, stateless TCP, FreeBSD

1. INTRODUCTION

A key challenge for HTTP-based content distributors is scaling their HTTP servers to handle growth in client-initiated load. The number of inbound connections per second may spike upwards for many different reasons – an active and successful marketing campaign, a denial of service attack, the so-called *Slashdot effect*, and so on. Traditional mitigation techniques include using load balancing to spread client-initiated TCP connections across identical servers, redirecting connections to caches geographically closer to individual clients (to both spread load and minimize the round trip times experienced by individual client connections), and simply adding more processing capacity to each web server.

In this paper we explore a server-side approach that can reduce the processing load per connection, and might conceivably be deployed alongside most existing load mitigation techniques. Our approach is built on a number of observations about client-initiated web traffic today.

Many websites are constructed from relatively small pages (blocks of HTML or related code) that themselves trigger the retrieval and display of many additional, yet also small, web objects. Thus the reply to a typical GET request is often a small number of Kbytes long. Because the maximum segment size (MSS) of TCP connections is often between 1400 and 1460 bytes (allowing for tunnels and other overhead along the IP path), such answers will fit within a handful of TCP/IP packets. Given TCP’s traditional “Initial Window” (IW) of three packets, and recent proposals to increase the IW to ten packets [2][1], a typical GET request is likely to be answered before TCP’s congestion control can play any role.

We suggest that TCP’s congestion control and reliability mechanisms play little part in the efficient transfer of small web objects in response to HTTP GET requests. In this paper we explore what benefits may be achieved by running web servers over a stripped-down transport protocol that looks like TCP ‘on the wire’ (so remote client TCP stacks remain happy), but keeps minimal state information within the server’s kernel. Our “Stateless” form of TCP (*statelessTCP*) for HTTP reduces the kernel resources required per TCP connection, and consequently increases the number of client-initiated HTTP connections per second a given combination of hardware and operating system can handle.

statelessTCP is only suitable for servers serving up small web objects. Sites with a mix of small and large web objects should structure their sites so that large objects are served by machines running a regular TCP stack.

The rest of this paper is structured as follows: after a discussion of related work in the next section, we present the design of statelessTCP for HTTP traffic in Section 3, and our test web server in Section 4. Then, in Section 5 we present some test results. Section 6 concludes.

2. RELATED WORK

The intention behind our work is to minimize a web server’s load. This is important when a server is receiving more requests than it can handle, such as when it is the victim of a Distributed Denial-of-Service (DDoS) attack. Because this is a severe problem and administrators of many important servers are so far still unable to cope with it (cf. the recent

[1]Such larger-than-the-standard-allows IW values are already being used in practice [16].

Pro-Wikileaks attacks on Visa and Mastercard), most of the related literature focuses on DDoS mitigation.

SYN cookies are a particularly well known method to reduce the load of a server that is being sent a lot of (fake) SYN packets. Normally, the server would be required to immediately create local state for each and every one of these packets – but with SYN cookies, the necessary information is only encoded in the server's response. This can be done by using the TCP header's sequence numbers [1] that the client reflects (increased by 1) in the acknowledgement field; this method shares with statelessTCP that it reduces the server's load with a server-only code change. However, statelessTCP reduces state for a whole connection, not just the SYN, and the server-side-only SYN cookie method also has some disadvantages (not being able to support TCP options, and lack of space in the sequence numbers). An extension to the TCP standard to solve these problems has recently been published [14]. SYN cookies are the best known, but not the only approach to mitigate SYN flooding attacks; a survey of known countermeasures is given in [3].

"Slowloris" [12] is an example of an attack that exploits server side state without having to send requests at a high rate; it forces a server to keep connections open by sending it partial HTTP requests on a regular basis. Common mitigation methods include limiting the number of concurrent connections per IP address, increasing the total number of concurrent connections allowed, and applying load balancers. statelessTCP cannot help against this attack because it does not address the issue that is being exploited: a vulnerability that is due to the way HTTP is specified.

"Not-A-Bot" (NAB), described in [6], attempts to detect the activity of humans as opposed to robots, akin to CAPTCHAs [18] but without requiring human involvement. Such automated systems can be used for a variety of purposes other than DDoS attack mitigation for web servers – e.g. spam detection and click-fraud mitigation. There is thus some overlap in the literature for these different purposes. Proposals such as NAB, the stateless TCP replacement "Trickles" [13] or the "Traffic Validation Architecture" (TVA) [19] however require a change to the client behavior, which is a key disadvantage over a scheme like statelessTCP which can improve a server's scaleability by only applying a local software update.

In contrast with our approach, none of the above overload mitigation strategies disable the server's congestion control. This has, however, been done for "normal" small web transfers for a long time now: Google has proposed using a larger initial window of 10 packets [2], and both Google and Microsoft were found to already apply this method even though it conflicts with the standard [16]. Even for hosts that use the standard initial window of 3 packets, a similar effect is attained by most web browsers, as they open a number of connections in parallel – the cumulative effect being the same as using a much larger initial window. statelessTCP can therefore be judged to be safe for deployment provided that its maximum file size is limited to at least 3 (conservative tuning) and at most 10 (progressive tuning) packets.

What we term 'statelessTCP' was originally[2] proposed (somewhat tongue-in-cheek) by Geoff Huston as a means to mitigate a potential performance issue faced by DNS servers

[2][17] describes an older, but very vague description of a roughly similar idea – albeit for a special "simple document server" instead of a standard service such as HTTP or DNS.

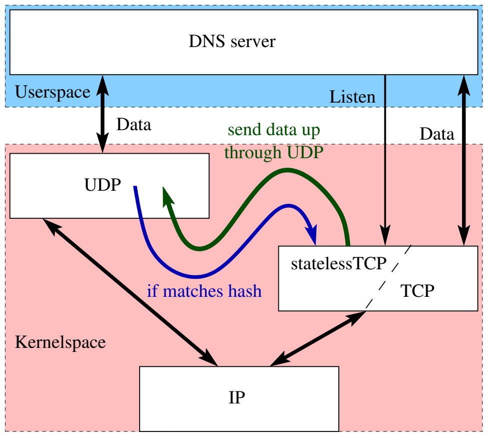


Figure 1: TCP as UDP

seeing a rise in DNS queries arriving over TCP. (The theory being that as more and more requests are expected to use IPv6 and DNSSEC, UDP responses will often exceed the commonly configured 512-byte packet size limit, requiring clients to switch to TCP for DNS queries [8].) We subsequently re-implemented this idea directly in the FreeBSD kernel and demonstrated significant potential performance improvements for DNS servers handling 100s of thousands of queries per second [7]. We have now leveraged our prior work to explore the potential of statelessTCP to support servers handling thousands of small HTTP transactions too.

3. "STATELESS" TCP FOR HTTP

Algorithms 1 and 2 show the basic operation of statelessTCP; it consists of a few simple rules for reacting to incoming control packets, and some additional behavior for dealing with application data. A timer ensures that we close the connection if the other side does not; we will explain in Section 3.3 why a RST flag is included in all FIN packets.

We translate the packets completely in kernel-space, passing only UDP packets with HTTP commands to the web sever which runs in user-space. The same applies for UDP packets generated by the web server – the translation to TCP occurs completely in kernel-space. Figure 1 illustrates the basic architecture.

The operation of statelessTCP is application-specific, in that it assumes a certain transaction sequence of the application protocol – algorithm 2 would never become active unless information from a previous incoming packet is available. This matches the behavior of HTTP, where a web server responds to incoming HTTP requests (typically HTTP GET commands). While the necessity to store some information prohibits statelessTCP from being strictly stateless, this is a very small amount of data compared to the state requirements of "normal" TCP – statelessTCP only uses the already existing SYN cache (also for its timer), and it neither requires a per-connection socket nor the `in_pcb` and `tcb` data structures. Packets are not buffered for retransmission. We implemented statelessTCP in FreeBSD 9.

3.1 statelessTCP and HTTP 1.0

Initially we tested statelessTCP using our test web server (see section 4) accepting only HTTP 1.0 type connections (no persistent connections). We installed it together with

Algorithm 1 statelessTCP algorithm outline: incoming

```
Capture every TCP packet arriving at port 80 and perform
the following operations, using a raw socket:

  1. If it is a TCP SYN packet, send back a TCP SYN/ACK
  2. If it is a TCP DATA packet,
     (a) store the acknowledgement number (and some more
         information) of the incoming packet and start a timer
     (b) send a TCP ACK packet back to the sender carrying no
         payload, and correct sequence and acknowledgement numbers
         based on the incoming packet's header
     (c) send the TCP payload to the server via a UDP socket
  3. If it is a TCP FIN packet, send back a TCP FIN/ACK/RST and
     remove the data stored at 2a, including the timer.
  4. If it is none of the above packet types, drop the packet
```

Algorithm 2 statelessTCP algorithm outline: outgoing

```
Event 1: upon a local UDP socket send request at port 80,
perform the following operations, using a raw socket:

  1. Check whether the source and destination addresses used in
     the socket call match any of the information stored at
     incoming/2a. If they do not match, let UDP carry on normal
     processing; otherwise continue:
  2. Depending on the size of the response, create one or
     multiple TCP packets:
     (a) For the first TCP packet, set the sequence number to the
         value stored at incoming/2a
     (b) For each additional packet, add the size of the previous
         packets payload to the previous packets sequence number
         and set the resulting value as the sequence number

Event 2: if the timer from incoming/2a fires, perform the
following operations, using a raw socket:

  1. Send a TCP FIN/RST to the host and port identified via the
     information that was stored with the timer, and remove the
     data stored at incoming/2a, including the timer.
```

statelessTCP on a FreeBSD desktop PC in our lab's LAN and accessed it from other PCs in the same LAN using the browsers listed above. We found that it works without any problems, using some static web pages containing small images. No packets were lost in this setup. Consistently, all web browsers requested HTTP 1.1, and, because our server announced HTTP 1.0, the browsers submitted a HTTP GET request for every object. These requests were actively terminated by the browsers, i.e. a FIN packet was sent by the client. This was accordingly answered with a TCP FIN/ACK/RST by statelessTCP (the RST played no role in these tests).

3.2 statelessTCP and HTTP 1.1

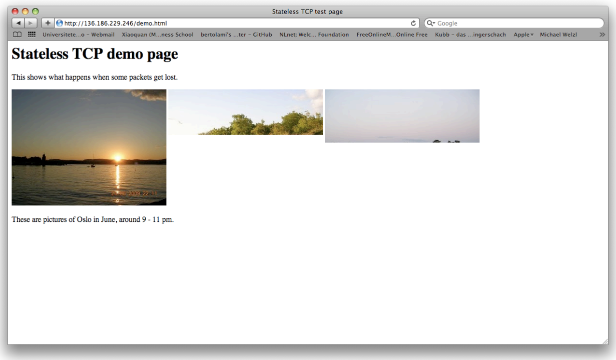
Persistent connections, which were introduced in HTTP 1.1, allow a web browser to issue multiple consecutive requests over a single TCP connection, thereby eliminating the connection setup/teardown overhead and delay. Restricting a web server to HTTP 1.0 is therefore at the cost of efficiency for the client. With HTTP 1.1, a connection can theoretically be closed by either side after a timeout [4]. However, we found that, if we let our web server announce HTTP 1.1 with persistent connections, browsers generally assume the server will close the connection after a while. This strikes us as an obvious strategy: keeping a connection available as long as possible for future use is in the interest of the browser, whereas it is in the interest of the server to close connections after a while in order to minimize the amount of state that is kept. This indeed matches the behavior exhibited by common web servers such as Apache.

An early version of statelessTCP did not include the timer that is described in Algorithms 1 and 2. With this version, persistent connections worked, but they were never closed. This resulted in the browser indicating ongoing download activity (the visualization depends on the browser, e.g. Firefox showed an animated symbol on the left side of the web page tab). We therefore needed to change statelessTCP to close the connection after a timeout. Luckily, the TCP SYN cache already includes a timer for every entry, which is normally used to issue another SYN/ACK in case it was never responded to. We reuse it to send a FIN/RST instead, which is correctly answered with a FIN/ACK by the client, causing the transaction to be terminated on both sides at the application level (i.e. the browser animation stops). Again, the RST played no role.

3.3 statelessTCP and packet loss

Since data is never retransmitted by statelessTCP, packet loss in the network can become a problem. As previously noted, statelessTCP is primarily for servers that must remain up under heavy loads; under such conditions, it allows access to web sites that could otherwise not be seen, and the risk of packet loss with its detrimental effects on the client side is outweighed by the ability to sometimes show a page at all. Nevertheless, we strived to minimize the problems caused by packet loss. To see what happens to a web client when packets are lost, we used dummynet to create various amounts of random loss on our web server. We saw that browsers generally display as much as they receive – but here, subtle differences between various browsers became evident. For example, Internet Explorer never showed images that were only partially available (instead, it showed a textual placeholder), whereas missing packets that make up a part of an image caused Firefox to partially display the image (with lines missing at the bottom). Figure 2 shows a screenshot of Safari displaying a page when some packets were lost.

Since a packet loss before a FIN causes the client's TCP to wait until any missing intermediate packets are received before it processes the FIN, we saw the browsers indicate ongoing download activity until the browser's "stop" button was pressed in our first tests. This was solved for most browsers by changing statelessTCP to always send a RST when it



Stateless TCP demo page

This shows what happens when some packets get lost.

These are pictures of Oslo in June, around 9 - 11 pm.

Figure 2: Safari displaying a page with packet loss

sends a FIN, and this never created any additional problems. If the lost packet was a HTML page, the FIN/RST caused browsers to display an error message (e.g. "The connection to the server was reset while the page was loading." in case of Firefox or "Safari can't open the page [url] because the server unexpectedly dropped the connection." in case of Safari), but if the browser was only waiting for an image or parts of an image, the download terminated. This seems to be a more appropriate behavior to us than showing that the download is still active when it really is not, but for service providers who disagree with this view, including RST flags on FINs could easily be made a configurable option.

4. TEST WEB SERVER (HTTPUDPD)

A web server operating with statelessTCP opens both TCP and UDP sockets to listen for requests. Client-initiated web requests received on the external TCP port are then handled by statelessTCP in the kernel. statelessTCP responds to TCP control packets, but sends data packets to the application via the UDP socket, created at initialization by the web server. It is assumed that the web server issues only one UDP transmission in response to the HTTP request. If it tries to send more data than would fit into a PathMTU-sized TCP packet, statelessTCP will (using the MTU information cached in the operating system) fragment it into multiple TCP packets. The total length of this data is limited by the maximum UDP packet length, which is 65507 byte – assuming the aforementioned common TCP MSS of 1460 byte, this would mean sending 45 packets, which is well beyond the number of packets that would be acceptable to send without doing congestion control. Thus, the UDP message length imposes no limitation on statelessTCP (but the maximum accepted amount of data should be set accordingly for the socket, e.g. via the `net.inet.udp.maxdgram` variable in case of FreeBSD).

To test statelessTCP we developed "httpudpd" (based upon "micro_httpd" by Jef Poskanzer)[3], and a variety of browsers (the latest versions of Firefox, Safari, Internet Explorer, Opera and Chrome as well as Gnu wget). micro_httpd is an exercise in creating the smallest possible web server in C – it is designed to run from inetd, i.e. it does not need

[3]statelessTCP, including httpudpd, is available from http://caia.swin.edu.au/ngen/statelesstcp/http and micro_httpd is available from http://acme.com/software/micro_httpd/

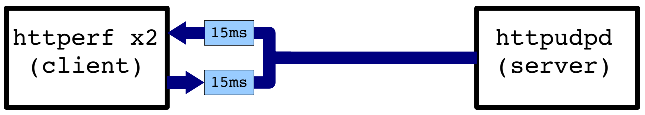


Figure 3: Test bed setup

to explicitly support parallel transactions and uses the stdin and stdout streams to communicate with a TCP socket. It has 286 lines of code (including comments and empty lines). We enhanced micro_httpd in several ways, yielding 520 lines of code:

- It listens on TCP, or TCP and UDP ports.
- It starts a fixed number of processes with `fork()` at initialization to handle parallel incoming http requests and avoid performance loss due to IOwaits.
- For TCP, it uses `select` with FreeBSD's accept filtering[4] [11, 10] (similar to Apache) to provide high performance HTTP request handling.
- For UDP, it uses `select`.
- It supports HTTP 1.1 with UDP.
- Apart from socket reads and writes, processing is identical for TCP and UDP.

5. PERFORMANCE TESTS

A simple testbed, shown in Figure 3, was used to test the relative performance of TCP and statelessTCP. Two instances of httperf [9] run on the (Linux) client, which also emulates a 15 ms delay in both directions using netem [5]. The client and server are connected on a 100 Mbps link through a switch. httpudpd (see Section 4) and statelessTCP run on the (FreeBSD) server. httpudpd starts 10 processes at initialization to handle incoming requests. The FreeBSD server runs with default parameters with the following two exceptions:

- the maximum UDP datagram size is set to 64000 B (net.inet.udp.maxdgram=64000),
- and the TCP SYN cache size is set to 2048 B. (net.inet.tcp.syncache.hashsize=2048)

The httperf clients request a single file from the httpudpd web server using HTTP 1.0 (non-persistent connections). Requests are made by each httperf instance at exponentially distributed random intervals (the request rate has a Poisson distribution). We measure:

- the average http request rate as determined by httperf,
- the load of the kernel process,
- the load of the httpudpd process[5],
- and the memory used by the kernel zone allocator [15].

The load of the kernel and httpudpd processes is measured using `ps -x`. The accumulated CPU time given by ps is sampled every 20 s, (21 times in total) with the difference between the samples used to plot the graphs. The

[4]Incoming TCP connections are accepted by the kernel, and only presented to the application program when the first data packet arrives

[5]The accumulated CPU time given by ps for a user process includes user + system time. Therefore the httpudpd measurement will include a portion of the kernel process CPU

Zone	Description
`socket`	memory used by the UNIX sockets.
`tcp_inpcb`	memory used to store network layer state (IP) for TCP connections.
`tcpcb`	memory used for the TCP control block.
`tcptw`	memory used to store the compressed TCP information for connections int the Time-Wait state
`syncache`	memory used in the TCP SYN cache

Table 1: Monitored memory use

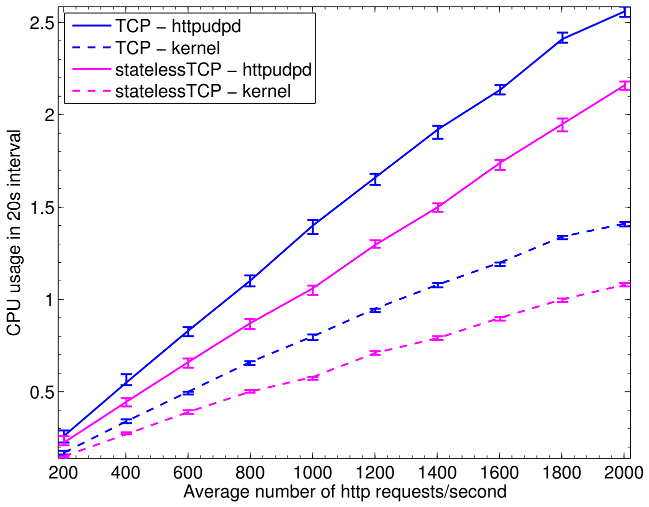


Figure 4: Httpudpd and Kernel load measured in accumulated CPU time in 20 s intervals

memory usage is sampled every 10 s (42 times in total) using `vmstat -z` to compare the average memory used in the communication process by statelessTCP and TCP. Specifically we look at the memory used by `socket`, `tcp_inpcb`, `tcpcb`, `tcptw`, and `syncache`.

Experiments are run with httperf parameterized for average http request rates of 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, and 2000 requests per second. Results are plotted against the achieved request rate as reported by httperf. Graphs show the 10th, 50th, and 90th percentiles (marker at the median, and error bars spanning the 10th to 90th percentiles).

5.1 Results

The relative loads for the http server are shown in Figure 4 for both normal TCP and statelessTCP modes. The httpudpd load is shown with a solid line, and the kernel load is shown with a dashed line. statelessTCP provides benefits for both kernel processing and the httpudpd process. At the higher loads, where the performance difference stabilizes, there is a 18–20 % improvement in the load due to the httpudpd process, and a 15–18 % improvement in the kernel process performance.

Memory consumption is another important issue for highly loaded web servers. Figure 5 shows the sampled memory usage for the memory zones outlined in table 1. Notice that `tcptw` does not increase with load in the both the TCP and statelessTCP plots (Figures 5(a) and 5(b)). After an IP address / port number combination has been used, it is placed in a TIME-WAIT state which prevents this combination from being used again for a period of time. FreeBSD has a default limit of 5150 previous connections whose state is stored in this way. In the normal TCP test, this limit is reached and maintained at all of the rates tested, resulting in the static memory use shown for these two elements. For statelessTCP, the very small amount of TIME-WAIT induced state is due to the test management script's interaction with the http server, not the HTTP requests. statelessTCP does not create and maintain full TCP state, so connections do not enter the TIME-WAIT state.

For the TCP tests, `tcpcb` and `socket` (and to a lesser extent `syncache`) increase with load, both containing the state necessary for active connections. `Tcpcb_inpcb` also does, though it is masked by the large proportion of `tcpcb_inpcb` state due to connections in the TIME-WAIT state.

Figure 5(b) shows that `syncache` dominates the statelessTCP memory usage.[6] Note that the graph scale is an order of magnitude smaller than in the TCP graph. statelessTCP extends the life of the SYN cache entries to enable the response to be returned to the client. `Syncache` entries use 144 B of memory, compared with 880 B for `tcpcb`.

A comparison of the total connection related memory usage is shown in Figure 5(c) with a two scale split y-axis. TCP's memory usage is in the order of 10^6 B, while statelessTCP's memory usage in the order of 10^4 B. Even without the huge static memory used by TCP due to the TIME-WAIT state, statelessTCP's memory increases much more slowly than TCP's memory. TCP requires `tcpcb` (880 B) + `tcpcb_inpcb` (336 B) + `socket` (680 B) for active connections, compared to statelessTCP's `syncache` (144 B) requirement.

6. CONCLUSION

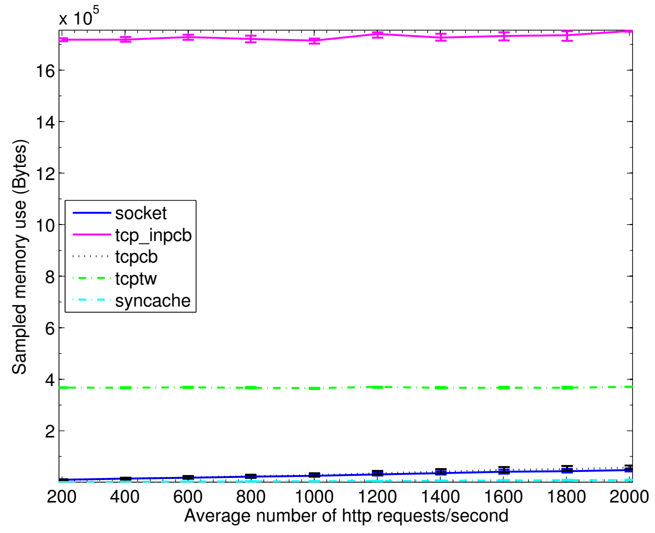
We have introduced statelessTCP and shown that it greatly reduces the CPU usage (> 20 %) and TCP related memory requirements (> 90 %) of web servers hosting very small files. It could be used to provide access to pages that could not normally be shown within a reasonable time frame, e.g. when a server is a target of a DDoS attack or when flash crowds occur, e.g. in emergency situations. As such, by removing the strict reliability requirement of TCP, it seems that statelessTCP can in fact increase the overall reliability of the web – but further work is needed to study the behavior of a more realistic web server (e.g. Apache) under extreme loads.

We have first applied statelessTCP to DNS and then made it work with HTTP with only a handful of minor changes; this leads us to believe that statelessTCP could just as easily be used in support of any other transactional application protocol.

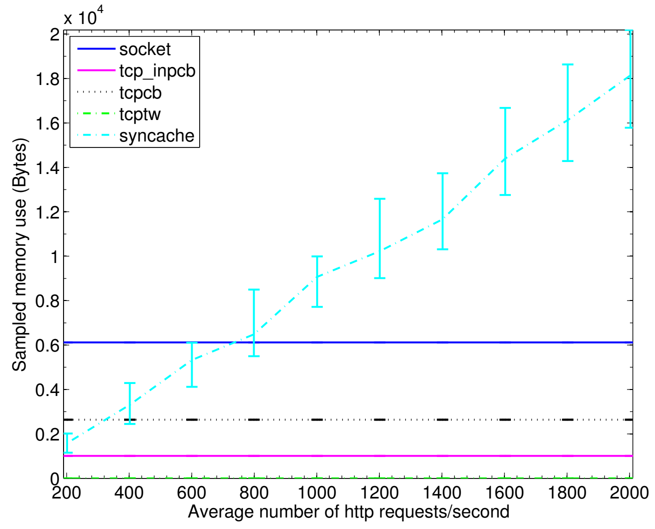
Acknowledgment

This work has evolved from earlier work made possible in part by grants from APNIC Pty Ltd and Nominet UK, and subsequent valuable discussions with Geoff Huston. The work has also been supported in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley.

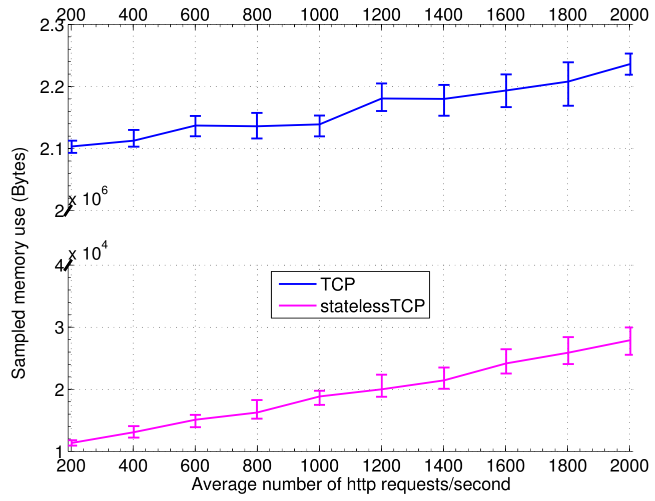
[6]The `tcpcb`, `tcpcb_inpcb` and `tcptw` memory consumption for statelessTCP is an artifact of the testing mechanism.



(a) TCP



(b) statelessTCP



(c) Combined memory usage

Figure 5: Sampled memory usage for statelessTCP

7. REFERENCES

[1] D. Bernstein. SYN cookies. http://cr.yp.to/syncookies.html.

[2] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing TCP's initial congestion window. *ACM SIGCOMM CCR*, 40:26–33, June 2010.

[3] W. Eddy. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational), Aug. 2007.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[5] L. Foundation. Netem — network emulation. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, Nov. 2009.

[6] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-bot: improving service availability in the face of botnet attacks. In *USENIX NSDI*, pages 307–320, Berkeley, CA, USA, 2009.

[7] D. Hayes, M. Rossi, and G. Armitage. Improving DNS performance using "Stateless" TCP in FreeBSD 9. Technical Report 101022A, Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, 22 October 2010.

[8] G. Huston. Stateless and dnsperate! The ISP Column, http://www.potaroo.net/ispcol/2009-11/stateless.pdf, November 2009.

[9] D. Mosberger, T. Jin, S. Eranian, and D. Carter. Httperf homepage. http://www.hpl.hp.com/research/linux/httperf/, Feb. 2009.

[10] A. Perlstein. ACCF_DATA — buffer incoming connections until data arrives. FreeBSD Man 9, Nov. 2000.

[11] A. Perlstein, S. Hearn, and J. R. van der Werven. ACCEPT_FILTER — filter incoming connections. FreeBSD Man 9, June 2000.

[12] RSnake. Slowloris. http://ha.ckers.org/slowloris/, June 2009.

[13] A. Shieh, A. C. Myers, and E. G. Sirer. A stateless approach to connection-oriented protocols. *ACM Trans. Comput. Syst.*, 26:8:1–8:50, September 2008.

[14] W. Simpson. TCP Cookie Transactions (TCPCT). RFC 6013 (Experimental), Jan. 2011.

[15] D.-E. Smorgrav and J. R. van der Werven. ZONE — zone allocator. FreeBSD Man 9, Oct. 2010.

[16] B. Strong. Google and Microsoft cheat on slow-start. Should you? http://blog.benstrong.com/2010/11/google-and-microsoft-cheat-on-slow.html, November 2010.

[17] supercat. Stateless tcp/ip server. http://www.halfbakery.com/idea/Stateless_20TCP_2fIP_20server, October 2004.

[18] L. von Ahn, M. Blum, N. Hopper, and J. Langford. Captcha: Using hard AI problems for security. In *EUROCRYPT 2003*. Springer, 2003.

[19] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM '05*, pages 241–252, New York, NY, USA, 2005. ACM.