# Tuning and Testing the FreeBSD 6 TCP Stack

Lawrence Stewart, James Healy Centre for Advanced Internet Architectures, Technical Report 070717B Swinburne University of Technology Melbourne, Australia lastewart@swin.edu.au, jhealy@swin.edu.au

Abstract—Tuning an operating system's network stack is crucial in order to achieve optimum performance from network intensive applications. This report discusses some of the key factors that affect network performance of FreeBSD 6.x based hosts. It then goes on to explain how to tune the FreeBSD 6.x network stack and how to easily test and evaluate the changes made.

*Index Terms*—FreeBSD, TCP, Tuning, Networking, Performance

#### I. INTRODUCTION

The widely used FreeBSD UNIX-like operating system provides a mature, stable and customisable platform, suitable for many tasks including hosting network intensive applications and telecommunications research. The operating system's network stack plays one of the most important roles in internetworking, as it is the avenue through which all packets enter and exit a device on the network.

Current TCP/IP stacks have evolved along with the various incremental additional standards documents that have appeared over the years, further increasing the complexity of an already sophisticated part of the operating system. Most of the standards have been optional additions to the protocol stack, and have been implemented as such in most operating systems.

CAIA's NewTCP project [1] mandated the use of the FreeBSD [2] operating system as the platform for performing our TCP research. As such, we performed some investigative research into how the FreeBSD TCP/IP stack operates. This report aims to capture the information learnt during this process for the benefit of future experimental TCP research using FreeBSD.

Whilst FreeBSD 6.2-RELEASE was used for the FreeBSD TCP/IP stack research we undertook, most of the technical information in this report will be applicable to all FreeBSD 6.x releases, and possibly to earlier (5.x and 4.x to a lesser extent) or up and coming (7.x and beyond) releases.

#### II. CONFIGURABLES

### A. The TCP Host Cache

The host cache is used to cache connection details and metrics to improve future performance of connections between the same hosts. At the completion of a TCP connection, a host will cache information for the connection for some defined period of time. If a new connection between the same 2 hosts is initiated before the cache has expired the entry, the connection will use the cached connection details to "seed" the connection's internal variables. This allows the connection to reach optimal performance significantly faster, as TCP does not need to go through its usual steps of learning what the optimal parameters for the connection are.

The cached details include the Round Trip Time (RTT) estimate to the remote host, path Maximum Transmission Unit (MTU) slow start threshold, congestion window and bandwidth-delay product (BDP). The hc\_metrics struct found in <netinet/tcp\_hostcache.c> lists the full set of connection details stored in the host cache.

The FreeBSD sysctl variables that affect the TCP host cache and their default values are shown in Listing 1.

#### Listing 1

6
net.inet.tcp.hostcache.cachelimit: 15360
net.inet.tcp.hostcache.hashsize: 512
net.inet.tcp.hostcache.bucketlimit: 30
net.inet.tcp.hostcache.count: 4
net.inet.tcp.hostcache.expire: 3600
net.inet.tcp.hostcache.purge: 0

*net.inet.tcp.hostcache.cachelimit* specifies the maximum number of cache entries that can be stored in the host cache. It is calculated as the product of the *net.inet.tcp.hostcache.hashsize* and *net.inet.tcp.hostcache.bucketlimit* variables. If the host cache ever reaches capacity, it simply begins overwriting

previous entries, which is not detrimental to the working of the system at all.

*net.inet.tcp.hostcache.count* lists the number of entries currently in the host cache.

*net.inet.tcp.hostcache.expire* lists the default time out value (in seconds) for cache entries. After initial creation, a cache entry will expire *net.inet.tcp.hostcache.expire* seconds after creation if a new connection to the same host is not made before the expiration. If a new connection is made, the corresponding cache entry's expiry timer is updated as well.

*net.inet.tcp.hostcache.purge* will force all current entries in the host cache to be removed next time the cache is pruned. After the prune is completed the value will be reset to 0.

In FreeBSD, the host cache is checked for stale entries every 5 minutes by default. This value can be modified by changing the <netinet/tcp\_hostcache.c> source file and recompiling the kernel. The "TCP\_HOSTCACHE\_PRUNE" define specifies the period between checks for stale cache entries.

Under normal circumstances, the default of 5 minutes is fine. However, when doing experimental research, we ideally limit or remove the affect of variables other than the one being studied. Given that cached connection settings can alter the dynamics of a TCP connection, it is advisable to nullify the effects of the host cache. This can be easily achieved by changing "TCP\_HOSTCACHE\_PRUNE" to, for example, 5 seconds, recompiling the kernel, and then setting *net.inet.tcp.hostcache.expire* to 1.

This will ensure all cache entries are removed within 5 seconds of the cache entry being created at the termination of the connection. It is then simply a matter of ensuring you wait at least 5 seconds between test runs to ensure the host cache is not affecting results. If this is too long a wait, you can decrease the time between checks even further.

During our investigation of the affects of the host cache on connections, we uncovered a quirk in the FreeBSD 6.2-RELEASE TCP stack. When a connection is made to a new host (one for which no host cache entry exists), and the RFC3390 TCP configuration option is enabled (see section II-B), the initial congestion window should be, and is in fact set to 4380 bytes, as recommended in RFC3390. However, when a host cache entry does exist, the initial congestion window is set by a different piece of code.

This piece of code is supposed to use the congestion window value from the host cache to seed the congestion window for the new connection. However, we discovered this does not occur correctly due to an incorrect assumption in the code, resulting in the congestion window always being set to 1 segment. The discussion on the FreeBSD "net" mailing list regarding this matter is archived here [3]. The end result of this is that use of the host cache for seeding the initial congestion window actually degrades the performance of subsequent connections between hosts, and obscures the intended behaviour of RFC3390. It is therefore advised that changing the value of "TCP\_HOSTCACHE\_PRUNE" and *net.inet.tcp.hostcache.expire* as previously described be carried out to mitigate the quirk, in situations where this may be problematic e.g. experimental research.

Note that setting the *net.inet.tcp.hostcache.cachelimit* tuner variable to 0 does not disable the host cache, and instead causes the kernel to panic whenever it goes to add a cache entry.

# B. TCP Extensions

A range of additional extensions to TCP have been proposed since the initial TCP standard was released. RFC4614 [4] provides an excellent summary of these various proposals and references to the various Request For Comment (RFC) documents that propose them.

For the extensions supported by FreeBSD, the sysctl interface [5] provides a way of enabling or disabling the extension and any applicable configuration variables. For the context of the following discussion unless explicitly stated otherwise, enabling a sysctl variable means setting the variable to a value of "1", and disabling means setting the variable to a value of "0". For example, to enable the TCP SACK extension, you would run the following command in a shell:

sysctl net.inet.tcp.sack.enable=1

The most relevant FreeBSD sysctl variables that affect the TCP stack and their default values are shown in Listing 2.

The full range of sysctl variables affecting FreeBSD's TCP stack can be found by running the following command in a shell:

#### sysctl -a | grep tcp

*net.inet.tcp.sack.enable* enables selective acknowledgments [6], [7] and [8]. Enabling SACK does not guarantee it will be used with all TCP connections, as it is a negotiated option that both sides must support. It is advised that this variable be enabled.

*net.inet.tcp.rfc1323* enables support for the window scaling TCP extension [9]. This should be enabled to facilitate high bandwidth transfers. As with SACK, this

# Listing 2

is a negotiated option and will only be utilised if both sides of the connection support it. It is advised that this variable be enabled.

*net.inet.tcp.rfc3042* enables support for the limited transmit mechanism [10]. This option is typically useful over lossy, small bandwidth-delay product paths to trigger fast recovery behaviour in circumstances where 3 duplicate ACKs have not been received. It is advised that this variable be enabled.

*net.inet.tcp.rfc3390* enables the stack to automatically select an initial congestion window size larger than 1 segment [11]. If this sysctl variable is not enabled, the sysctl variables *net.inet.tcp.slowstart\_flightsize* and *net.inet.tcp.local\_slowstart\_flightsize* can be used to manually control the initial size of the congestion window for both remote (off subnet) and local hosts respectively. The values are specified in multiples of the segment size. The rfc3390 option is enabled by default and should only be disabled if manual control of the initial congestion window is required.

Note that if the host cache contains an entry relevant to a new connection, the initial congestion window will be set as a function of the cache entry, not the three sysctl variables decribed above.

Enabling *net.inet.tcp.delayed\_ack* allows the host to hold off sending an acknolwedgement for a specified period of time, instead of immediately after receiving a TCP segment, as described in [12]. This allows the host to wait for more data to arrive, which it can then potentially send a combined acknowledgement for.

The usefullness of this variable diminishes as the RTT between communicating hosts decreases, to the point where use of delayed ACKs can actually cause harm to connection performance in some rare situations. Connections over networks with sub 10ms RTTs can experience reduced throughput as a result of delayed ACKing if non-zero packet loss is present. This behaviour occurs if the connection's congestion window ever becomes equivalent to 1 segment size. This can occur if there is no data in flight after exiting fast recovery mode or during initial startup of the connection. A congestion window of 1 segment will result in the sending host sending only 1 segment, which will not be ACKed by the receiver using delayed ACKs until the delayed ACK timer expires. On a low RTT network, waiting for this timer to expire can dramatically reduce throughput on account of not being able to transmit data whilst waiting for the ACK.

For the majority of connection scenarios though, leaving this variable enabled will not cause significant performance issues, and it is therefore recommended it remain enabled.

The amount of time to wait before sending an acknowledgment when delayed ACKing is enabled is controlled by the *net.inet.tcp.delacktime* sysctl variable, which defaults to 100ms.

*net.inet.tcp.inflight.enable* enables support for BDP estimation and subsequent TCP window limiting based on the estimated BDP. Enabling this variable will cause FreeBSD to limit the amount of inflight data to the estimated path BDP. This actively attempts to avoid TCP saturating the path and falling into the cyclical behaviour of increasingly probing the path, backing off and repeating. In theory, this should allow better aggregate throughput. However, enabling this option will obscure the behaviour of the TCP congestion control mechanism "at work" and should probably be disabled for TCP research purposes, unless you know you have a specific need for this option.

The RTT above which the inflight mechanism will begin to affect TCP connections is controlled by the *net.inet.tcp.inflight.rttthresh* sysctl variable, and defaults to 10ms.

*net.inet.tcp.newreno* enables support for the new reno congestion control modification [13]. This improves the performance of a flow when multiple packets are lost in a single window. It is advised that this variable be enabled.

*net.inet.tcp.path\_mtu\_discovery* enables discovery of the maximum transmission unit, as described in [14]. It is advised that this variable be enabled.

# C. Buffers

A number of different buffers are used within the kernel for network stack related operations. Listing 3

shows the relevant loader tunables and sysctl variables that can be used to tune these buffers.

Listing 3	
kern.ipc.nmbclusters	
kern.ipc.maxsockbuf	
net.inet.tcp.sendspace	
net.inet.tcp.recvspace	

*kern.ipc.nmbclusters* defines the maximum number of mbuf clusters that can be in use by the network stack at any one time. Being a loader tunable, it must be set in /boot/loader.conf and requires a system restart in order to take effect. However, this variable is auto-tuned by FreeBSD if not explicitly overridden in /boot/loader.conf, and will be sufficiently sized for most situations. See [15] for more information.

*kern.ipc.maxsockbuf* defines the maximum size, in bytes, that can be allocated to a socket send or receive buffer i.e. it places an upper bound on the values *net.inet.tcp.sendspace* and *net.inet.tcp.recvspace* can take.

*net.inet.tcp.sendspace* defines the default buffer size, in bytes, for use with outbound data sent using a TCP socket. This variable must be set to a value smaller than *kern.ipc.maxsockbuf*. Applications can manually override the size of the send buffer when they create a socket using the setsockopt() system call. Tuning this variable will affect the host's ability to service TCP windows advertised by other hosts. For example, tuning this variable to 1MB will likely constrain our ability (subject to other parameters like bandwith, processing ability, etc.) to send data to a host that has advertised a window of 2MB. That is of course assuming the application has not explicitly overridden the send space buffer size using setsockopt().

*net.inet.tcp.recvspace* defines the default buffer size, in bytes, for use with inbound data received using a TCP socket. This variable effectively caps the maximum TCP window size the kernel will advertise to other hosts, and must be set to a value smaller than *kern.ipc.maxsockbuf*. Applications can manually override the size of the receive buffer when they create a socket using the setsockopt() system call. Tuning this variable will affect the host's ability to receive data from other hosts. For example, tuning this variable to 1MB will likely constrain our ability (subject to other parameters like bandwith, processing ability, etc.) to receive data from a host that is capable of servicing a window of 2MB. That is of course assuming the application has not explicitly overridden the receive space buffer size using setsockopt().

Tuning *kern.ipc.maxsockbuf*, *net.inet.tcp.sendspace* and *net.inet.tcp.recvspace* are of particular importance to ensuring optimal TCP performance. For example, limiting the receive space buffer, and therefore the maximum advertised TCP window, to 64KB in a gigabit Ethernet environment with 10ms of delay will artifically limit your TCP throughput to 52Mbps. It is therefore important to set these values appropriately for your situation, using BDP calculations relevant to your network to guide you.

# D. MTU

The underlying network interface's maximum transmission unit can have a significant impact on the efficiency and throughput of network traffic handling. Faster packet per second rates of smaller packets places a larger load on the system than lower packet per second rates of larger packets. This is typically only useful for communications between hosts on a common local area network, as jumbo frames across the Internet does not yet have widespread support.

Using an MTU greater than 1500 bytes requires a network interface card and driver that both support jumbo frames. Intel based gigabit ethernet cards using the "em" driver seem to have the best jumbo frame support in FreeBSD. We used PCI Intel PRO/1000 GT cards with a MTU of 16110 bytes without any problems. Based on some simple testing, we found that the optimum MTU was about 10000 bytes on a 2.8GHz P4, but this is likely to differ across hardware configurations.

Changing the MTU of an interface is simple. An example is shown in Listing 4. Specifying the network interface's IP address in the ifconfig command will ensure the routing table is updated to reflect the new MTU. This is important, as FreeBSD uses the routing table to determine which MTU it should use to communicate to a particular host with. In this way, a FreeBSD host capable of transmitting jumbo frames can coexist on a LAN with hosts that can only receive standard 1500 byte frames, by using its routing table to determine which frame size to use for a particular host. Running "netstat -i" will show you if your new MTU has been reflected in the routing table. You can manually add routes to control the MTU used for certain hosts using the "route" command [16].

Listing 4			
ifconfig em0	10.0.0.1	mtu 9000	

If your network interface card or driver do not support

CAIA Technical Report 070717B

jumbo frames, attempting to set an MTU greater than 1500 will result in ifconfig generating an error similar to: "ifconfig: ioctl (set mtu): Invalid argument".

### III. TESTING TOOLS

Numerous tools have been developed over the years to test various aspects of network performance. Here we describe a selection that we have used, and provide references to sources of additional information.

### A. Iperf

Of the tools we evaluated, Iperf [17] v2.0.2 has to be the easiest to use. It provides a full featured C++ based TCP and UDP network testing application that runs on a large range of platforms. Iperf uses a client and server to perform tests, with data being sent from client to server in typical unidirectional tests.

To perform a basic throughput test, install Iperf on two machines reachable via an IP network. On one machine, start an Iperf server, by running the command "iperf -s". After starting the server, start the Iperf client by running the command "iperf -c <server\_ip>", where <server\_ip> is the IP address of the machine running the Iperf server. This will commence a 10 second test, which will print the overall amount of data transferred and throughput achieved on completion.

There are a whole range of options to tweak, which can be found by running "iperf -h". Of particular interest for TCP testing are the "-F", "-t" and "-w" options. The "-F" option allows you to specify a data file to transmit as the payload, instead of using Iperf's default repeating number payload. The "-t" option allows you to specify the length of the test to run. The "-w" option allows you to specify the TCP window size used by Iperf.

Due to an oddity in Iperf v2.0.2, the "-w" switch when used in client mode does not actually affect the TCP window size, and instead modifies the send space buffer of the TCP socket rather than the receive space buffer. We have released a patch against the Iperf v2.0.2 source that addresses this issue. The patch can be downloaded from here [18], along with a readme explaining how to use it.

## B. nttcp

We also used nttcp [19] to validate the results Iperf was giving us. Similar to Iperf, nttcp uses a client and server to perform tests, with data being sent from client to server in typical unidirectional tests.

To perform a basic throughput test, install nttcp on two machines reachable via an IP network. On one machine, start a nttcp server, by running the command "nttcp i". After starting the server, start the nttcp client by running the command "nttcp -T <server\_ip>", where <server\_ip> is the IP address of the machine running the nttcp server. This will commence an untimed test that will complete when 8388608 bytes have been transferred. A test summary will be printed to screen on completion.

There are a whole range of options to tweak, which can be found by running "nttcp" without any arguments. Of particular interest for TCP testing are the "-w", "l" and "-n" options. The "-w" option allows you to specify the size of the send buffer used by the nttcp TCP socket. The "-l" option allows you to specify the size of the buffers transmitted by nttcp during the test. The "-n" option allows you to specify the number of buffers transmitted by nttcp during the test. The product of the buffer size (specified using "-l") and number of buffers (specified using "-n") equates to the number of bytes transmitted during the test.

# C. SIFTR

SIFTR [18] is a new tool we developed to give TCP researchers using FreeBSD access to kernel-level details from the network stack about active TCP connections. Using SIFTR in combination with a testing tool like Iperf or nttcp and a traffic capture utility like Tcpdump, you can gain access to almost every important piece of information required to see exactly what the test tool and operating system are doing and actually putting on the wire. You can also gain insights into the network stack's perception of the network the test flow is traversing, by having access to data like the kernel's RTT and bandwidth estimations for the flow.

## D. More info

The 3 tools we have described above are only a small fraction of those that exist. The following references should provide further avenues for learning about generic and FreeBSD specific information and tools related to TCP performance and research: [20] [21] [22] [23] [24].

## IV. CONCLUSION

Having outlined and discussed some of the key factors influencing TCP/IP performance in FreeBSD 6.x, you should now have a better understanding of TCP tuning in general and how to go about it on your own equipment.

#### V. ACKNOWLEDGMENTS

This report has been made possible in part by a grant from the Cisco University Research Program Fund at Community Foundation Silicon Valley.

#### REFERENCES

- "The NewTCP Project," June 2007, http://caia.swin.edu.au/urp/ newtcp.
- [2] "The FreeBSD Project," June 2007, http://www.freebsd.org/.
- [3] James Healy, "Odd congestion window behaviour," July 2007, http://lists.freebsd.org/pipermail/freebsd-net/2007-July/014781. html.
- [4] M. Duke, R. Braden, W. Eddy, E. Blanton, "RFC 4614: A Roadmap for Transmission Control Protocol (TCP) Specification Documents," September 2006, http://www.ietf.org/rfc/ rfc4614.txt.
- [5] FreeBSD Hypertext Man Pages, "SYSCTL," June 2007, http://www.freebsd.org/cgi/man.cgi?query=sysctl&sektion=8.
- [6] "RFC 2018: TCP Selective Acknowledgement Options," October 1996, http://www.ietf.org/rfc/rfc2018.txt.
- [7] "RFC 2883: An Extension to the Selective Acknowledgement (SACK) Option for TCP," July 2000, http://www.ietf.org/rfc/ rfc2883.txt.
- [8] "RFC 3517: A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP," April 2003, http://www.ietf.org/rfc/rfc3517.txt.
- [9] "RFC 1323: TCP Extensions for High Performance," May 1992, http://www.ietf.org/rfc/rfc1323.txt.
- [10] "RFC 3042: Enhancing TCP's Loss Recovery Using Limited Transmit," January 2001, http://www.ietf.org/rfc/rfc3042.txt.

- [11] "RFC 3390: Increasing TCP's Initial Window," October 2002, http://www.ietf.org/rfc/rfc3390.txt.
- [12] "RFC 813: Window and Acknowledgement Strategy in TCP," September 1982, http://www.ietf.org/rfc/rfc813.txt.
- [13] "RFC 3782: The NewReno Modification to TCP's Fast Recovery Algorithm," April 2004, http://www.ietf.org/rfc/rfc3782.txt.
- [14] "RFC 1191: Path MTU Discovery," November 1990, http: //www.ietf.org/rfc/rfc1191.txt.
- [15] The FreeBSD Project, "Tuning Kernel Limits," June 2007, http://www.freebsd.org/doc/en\_US.ISO8859-1/books/ handbook/configtuning-kernel-limits.html.
- [16] FreeBSD Hypertext Man Pages, "ROUTE," June 2007, http: //www.freebsd.org/cgi/man.cgi?query=route&sektion=8.
- [17] , "Iperf The TCP/UDP Bandwidth Measurement Tool," June 2007, http://dast.nlanr.net/Projects/Iperf/.
- [18] "NewTCP project tools," June 2007, http://caia.swin.edu.au/urp/ newtcp/tools.html.
- [19] Elmar Bartel, "NTTCP Source," July 2007, http://freeware.sgi. com/source/nttcp/.
- [20] Rick Jones, "Netperf Homepage," July 2007, http://www. netperf.org/.
- [21] Sally Floyd, "Transport Modeling Research Group (TMRG)," July 2007, http://www.icir.org/tmrg/.
- [22] S. Floyd, E. Kohler, "Tools for the Evaluation of Simulation and Testbed Scenarios," Internet Engineering Task Force, Tech. Rep., December 2006.
- [23] FreeBSD Hypertext Man Pages, "SYSTAT," July 2007, http://www.freebsd.org/cgi/man.cgi?query=systat&sektion=1.
- [24] —, "NETSTAT," July 2007, http://www.freebsd.org/cgi/man. cgi?query=netstat&sektion=1.