Fun with Dummynet: Maintaining high throughput in large bandwidth-delay product networks

Andy Jones andjones@cs.unc.edu

December 5th, 2006

Abstract

Maintaining high throughput in large bandwidth-delay product networks is widely known problem arising from TCP's additive increase, multiplicitive decrease (AIMD) algorithm. Several alternatives to AIMD have been proposed, some of the most popular being High-Speed TCP (HSTCP), Fast TCP, Scalable TCP (STCP), CUBIC, and HTCP. Evaluating how each of these alternatives operates on a real network (not simulated) helps us see how each will behave in the wild, against each other, and against current TCP implementations. This in turn helps us improve upon our work and bring us one step closer toward releasing high-speed TCP (in the general sense) for public consumption.

1 Introduction

My semester project involved three components:

- Implementing high-speed variants in the FreeBSD 6.0 kernel
- Porting Usernet to the FreeBSD 6.0 kernel
- Calibrate our test-bed and dummynet, which we use setup link characteristics

It is in the third item that I spent a gross majority of my time. The remainer of this technical report is devoted to the painful process of kernel debugging, the FreeBSD network stack, and howto maintain high throughput in large bandwidth-delay product networks.

Socket kern/uipc_socket.c
TCP Output netinet/tcp_output.c
IP Output netinet/ip_output.c
Dummynet netinet/ip_dummynet.c
IP Output netinet/ip_output.c
Device Hand-off net/if_ether.c
Device Driver dev/em/if_em.c
Physical Media

Figure 1: FreeBSD's network stack

This rest of this report is organized as follows. Section 2 talks about the FreeBSD network stack, giving pointers to code and documentation where appropriate. Section 3 talks about kernel hacking with the examples coming from the work done is Section 4 "What's wrong with Dummynet?". Section 5 gives a validation of the work done. The appendices have information that may be of interest to various readers.

2 Network Stack

Figure 1 illustrates the path a packet takes from the application's socket to the physical media in the FreeBSD kernel. The output path is shown, the input path is similiar, replace output with input where appropriate.

One thing to note is that IP Output is listed twice. This is intentional. The FreeBSD firewall is implemented as a series of hooks. A hook is simply a piece of code that is interested in receiving packets and possible processing them. Anyone can write a hook (called a pfil_hook : see net/pfil.c), the prototype looks like:

```
int packet_hook(void *arg, struct mbuf **m, struct ifnet *interface,
                int direction, struct inpcb *inp_control_block);
```

Figure 2: A Pfil Hook

A hook may eat a packet (processing stops) by returning 0, or let processing continue (return non-zero). The ipfw (IP Firewall) is one such hook that forwards packets to dummynet depending on the rules in ipfw. It is in dummynet where we can delay a packet, enforce bandwidth limitation, set loss probabilities, and more. In Figure 1, ip_output calls pfil_run_hooks, where dummynet will consume the packet, then dummynet will pass the packet back to ip_output when it is done delaying it, losing it, or bandwidth limiting it.

It is also in dummynet where we have the most problems trying to maintain a high throughput in a large bandwidth-delay product network. Before we get into dummynet though, let's discuss writing and debugging kernel code.

3 Kernel Hacking

This section exist purely to help any would be future kernel hacker get a jump start on kernel hacking. Many tools exist to debug programs we write in C, Java, or other languages. gdb for C allows us to step through programs, set break points, examine the stack, memory values and more. When an uncaught exception brings down the JVM, java is nice enough to print a stack trace. Many goto using code-slinging programmers like to use print statements to debug code. These are the two popular methods for debugging, however in the kernel, neither work.

Before we get ahead of ourselves, we should note that when a kernel crashes, it does dump memory and what it was most recently working on, and you may use print statements in your kernel. However, the utility of any information that either method gives is rather dubious at best. When one is trying to debug networking code, one might want to test a race condition that only occurs after ten-thousand packets fly by. Ten-thousand prints or ten-thousand steps is sub-optimal at best. Worse yet is when the condition occurs rarely, but often enough to muck up experimental data. The bug is hard to replicate, and test, compile, debug cycles are long.

To get in the right mind-set for kernel hacking:

- Forget all about floating point numbers
- Forget about using a debugger
- Forget about using print statements

That being said, get the fastest machine possible to reduce compile time, consider writing some scripts to help automate deploying new kernels and backing up old ones. Now, let's figure out how to get data in and out of the kernel from user-land. Sysctl variables are the key here. For the crash course, see Figure 3.

For more information on sysctl variables and their use and syntax, see the "How do I read/mess with sysctl variables" available on the DiRT FAQ.

```
// adding a variable in the kernel code
u_int tcp_cwnd = 0;

SYSCTL_UINT(_net_inet_tcp_congestion, OID_AUTO, cwnd, CTLFLAG_RW,
    &tcp_cwnd, 0, ``Size of congestion window (in packets)
                   for flow of last received packet'');

// getting at it from userland
// Read
% sysctl net.inet.tcp.congestion.tcp_cwnd
// Write
% sysctl net.inet.tcp.congestion.tcp_cwnd=52
// Search
% sysctl -a | grep net.inet.tcp | less
```

Figure 3: The crash course into sysctl variables

Given a sysctl variable, we can get data out of the kernel. That still begs the question, what are we measuring?

3.1 Spot Values

Even if the variable measured changes much faster than it can be measured, don't dismiss spot values. Spot values can be quite useful for monotonically increases sequences, or variables with small variance. One can measure the current congestion window this way. It is a monotonically increasing sequence (until a congestion event occurs) that may change 20,000 times per second, however we can be fairly confident about interpolated values knowing how TCP works. We could also use a spot value to measure the current queue size in dummynet. The queue size will vary very rapily as packets are queued and dequeued every scheduling tick, but its overall value will vary little from its true average. In this case, the spot value is probably close to the average.

3.2 Exponential Weighted Moving Average (EWMA)

Using a EWMA is appropriate when one wishes to measure a variable whose variance is high, outliers persist, or the nothing is known about trends in the data. EWMA is appropriate as it is essentially a low-pass filter that allows us to look at trends in time while disregarding outliers. The general formula for an EWMA is

$$x_{n+1} = \alpha x_n + x'(1 - \alpha)$$

where α depends on how fast the variable x changes. Typically a value of 0.99 [1] will be used. EWMA can come in handy, for instance, when trying to measure how much time a piece of kernel code takes (see appendix A for fine-granularity timing). A start and end time is marked, and their difference is used as x' in the equation to update x. This comes in handy when trying to time a piece of code who's running time may change wildly depending on the number of packets being processed, locking issues, or otherwise.

4 What's Wrong with Dummynet?

One subtle bug exists in the stock Dummynet implementation that should be corrected for experiments. When a packet arrives in dummynet it is shoved into a queue which limits the bandwidth a TCP flow may use. Upon exit from the queue, the pipe is transferred to a pipe where it sits for any configured amount of delay time and might possibly be dropped depending on the loss probability. Once the delay time has passed, the packet is released to ip_output. [2]

Dummynet time is in scheduling ticks. Dummynet, ideally, is called once per scheduling tick. Instead of keeping time in milli or micro-seconds, dummynet keeps time in ticks, or how many times it was called by the scheduler. All time (delay in milli-seconds or otherwise) is converted to ticks by dummynet. How many ticks? Depends on the frequency (HZ) set while compling the kernel. Should the dummynet output routine take longer than one scheduling tick (which is 1/Hz or 1ms for a 1000Hz clock), it will have missed a scheduling tick(s) and reschedule itself for the next scheduling tick, which may be long after the time it started depending on how many packets dummynet tried to flush. When this happens, the machine starts thrashing. All incoming ACKs that generate new packets for dummynet to process all have the same output time. This will cause dummynet to miss more and more ticks as the congestion window grows.

The solution is to have dummynet keep an absolute time instead of the relative time it uses. Figure 4 illustrates how to fix this problem.

[1]Since floating point number and floating point math cannot be used, α must be shifted to an integer and the result shifted back

[2]the idea of pipes and queues in the dummynet kernel code is slightly different from the pipes and queues as explained in the ipfw man page for dummynet

```
microuptime(&current_output);
// return time elapsed in micro-seconds
time_diff = TIME_DIFF(&start_time, &current_output);
// convert time to hz ticks
curr_time = time_diff / (1000000 / hz);
```

Figure 4: Modified dummynet code from the void dummynet(void *) routine

It is also easy to forget that dummynet is also queueing packets in the network. This can create increased delay and eventually a course-grained timeouts in the TCP code, especially when dealing with high bandwidth-delay product networks. This behavior is easy to see on a 1Gb network with a delay of 50ms and 12MB in send/receive space.

5 Getting that high throughput

The following steps can be taken to squeeze the highest throughput possible out of FreeBSD in high bandwidth-delay product networks.

5.1 Setting up the Network

First, we have to know how much buffer space we'll need. One way to think about the amount of buffer space TCP needs is to think about how much data needs to be transferred per round trip. If the RTT is 100ms, that leaves us 10 round trips per second. If we want to transfer 1Gb per second, we must transfer 100Mb per round trip or 100 / 8 = 12.5MB of data per round trip. More formally $B = S/RTT/8$ where B is the buffer space needed and S is the speed to acheive.

To allocate this much memory, the following commands can be used:

```
sysctl kern.ipc.maxsockbuf=‘echo ‘‘1048576 16 *p’’ | dc‘
sysctl net.inet.tcp.sendspace=‘echo ‘‘1048576 12 *p’’ | dc‘
sysctl net.inet.tcp.recvspace=‘echo ‘‘1048576 12 *p’’ | dc‘
```

This gives us a maximum socket buffer of 16MB, and a TCP send and receive buffer of 12MB, respectively. If you find yourself running out of memory, you may want to set the sysctl variable kern.ipc.nmbclusters higher.

Unfortunately, this variable must be set at boot time by placing the line "kern.ipc.nmbclusters=65536" in the file /boot/loader.conf [3]

Next we tune other sysctl variables:

```
// turn off inflight extension,
// it KILLS throughput with low variance flows
sysctl net.inet.tcp.inflight.enable=0
// TCP's large window extension
sysctl net.inet.tcp.rfc1323=1
// Use delayed acknowledgements
sysctl net.inet.tcp.delayed_ack=1
```

5.2 Setup Dummynet

Last, we setup the dummynet pipe:

```
ipfw pipe 1 config delay 50ms bw 1000Mbit/s
ipfw add 10 pipe 1 ip from any to any out
```

You'll next want to test ping any machine to determine if the setup is correct. After that, feel free to run a full blown iperf test.

```
// on the receiver side
[root@chile139]$ iperf -s

// on the sender side
[root@peru138]$ iperf -c chile139 -t 60 -i 1
```

5.3 Experiments

We now come to the last section which give a brief validation of the values and work above. Figures 5 shows how the throughput has dramatically improved for different values of delay. The throughput is not exactly where we'd like it, especially for higher delays (50ms and beyond), but it is significantly better than where we started.

Figures 6 through 8 show how the growth of the congestion window differs from stock TCP. We notice the more aggresive growth of each algorithm compared to stock TCP. Each test was conducted using a delay of 50ms, a 12MB send/receive buffer, and a packet loss rate of .0001. These experiments are short (only 2 minutes each), but they are a simple visual validation that the congestion window is growing correctly for each algorithm.

[3]You must be root to make all the changes listed here

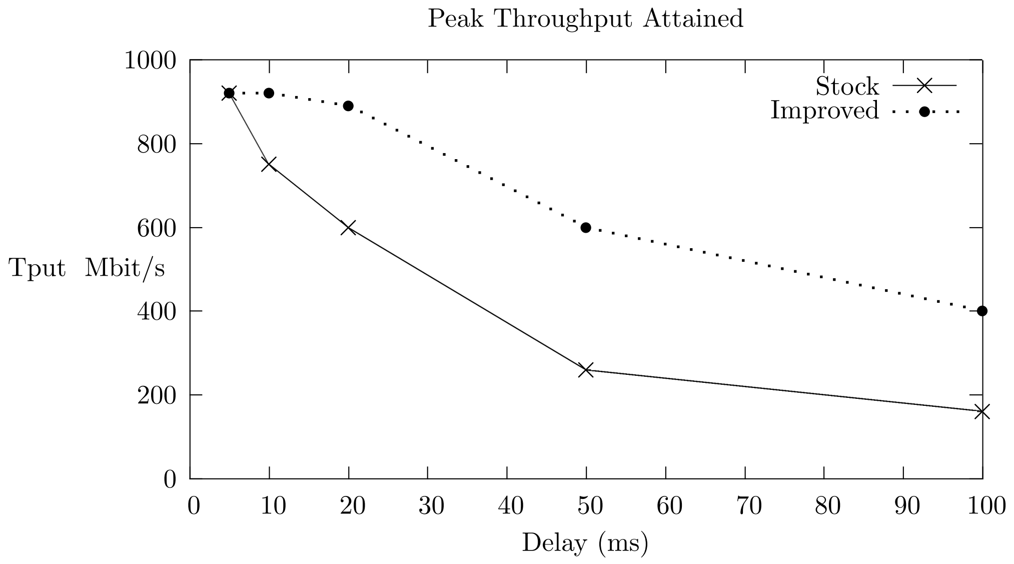

Figure 5: Standard TCP

Future work invovles running more thorough experiments in a wide range of parameters including varying delay, network load, network loss, and seeing how these protocols compete against each other and stock TCP. Dina Katabi's XCP is also available to us (see DiRT FAQ). A thorough testing of the parameter space for these 4 algorithms should provide us with clues about what works and what does not. Much of this work forms part of Jay Aikat's disseration.

Appendix A: Fine-granularity timing

The code in Figure A will grab the instruction counter (from an x86 architecture) and throw it into variable x. The instruction counter (as the name implies) increments once per instruction and is the finest granularity timer on the chip. The number of instructions per second obviously varies depending on the processor installed. Consequently we have to work backwards to find the number of instructions per second to relate I/S back to time and have data with time units (much preferred).

Here's how to figure out how many ticks per second there are:

- Grab the start time
- Grab the starting instruction counter

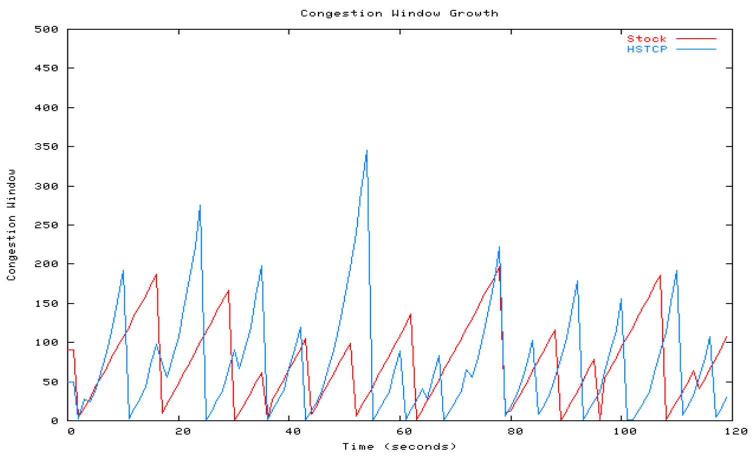

Figure 6: Stock TCP versus HSTCP

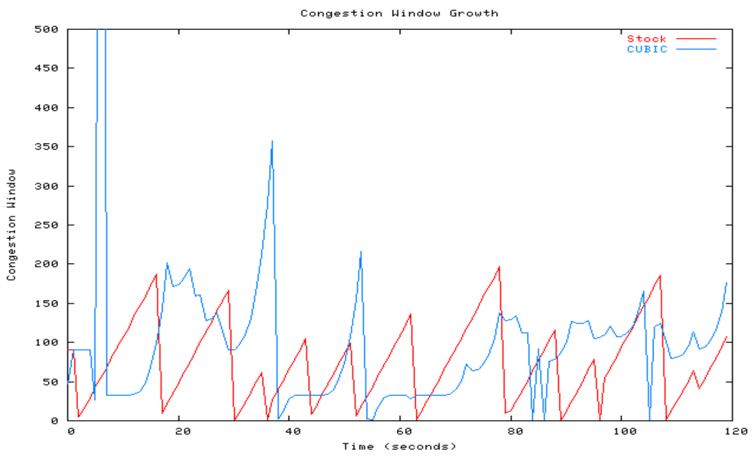

Figure 7: Stock TCP versus CUBIC

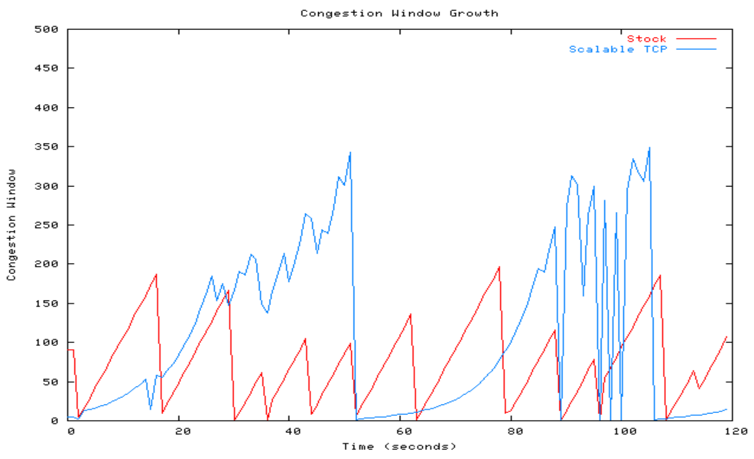

Figure 8: Stock TCP versus STCP

```
__inline__ u_int64_t rdtsc(void * notused) {
  u_int64_t x;
  __asm__ volatile (``.byte 0x0f, 0x31'' : ``=A'' (x));
  return x;
}
```

Figure A

- Do something time intensive
- Grab the end time
- Grab the ending instruction counter
- Subtract, Divide

Appendix B: Implementing Congestion Algorithms

The more relevant places where TCP congestion control takes place, all in netinet/

tcp_timer.c	Course-grained timeouts happen here (near bottom of file)
tcp_input.c	ACKs are received here. You'll want to look for the triple duplicate ACK code and where they linearly open the congestion window
tcp_var.h	TCP control block is here. If you want to add per-flow state information, this is the place to do it
tcp_subr.c	TCP init happens here. If you want to initialize per-flow state, do it here

Appendix C: Sysctl Variables for HSTCP

- net.inet.tcp.congestion.impl - current congestion algorithm uses values 0-5 (see table below)
- net.inet.tcp.congestion.cwnd - current size of congestion window (packets)
- ipfw pipe 1 config delay -1 src port
- ipfw pipe 1 config delay -2 dst port
- ipfw pipe 1 config delay -10 delay from distribution (see netinet/dummynet_table.h)

Fast TCP and HTCP are not implemented as of the time of this writing (December 6th, 2006). Values of net.inet.tcp.congestion.impl:

Value	Algorithm
0	Normal TCP
1	HSTCP
2	CUBIC
3	STCP
4	Fast TCP
5	HTCP