Evaluating the FreeBSD 9.x Modular Congestion Control Framework's Performance Impact

David Hayes, Lawrence Stewart, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 110228A
Swinburne University of Technology
Melbourne, Australia
dahayes@swin.edu.au, lastewart@swin.edu.au, garmitage@swin.edu.au

***Abstract*—Modular congestion control (modCC) is a recently added feature to the FreeBSD kernel. The implementation introduced a number of key changes to the TCP stack with the potential to impact performance. We test the relative performance of TCP before and after these changes using the 215163 and 217806 Subversion repository revisions of FreeBSD's "head" (9.x) development branch respectively. We find that the modCC changes do not adversely impact performance on the whole, and in fact slightly improve the nett performance of the FreeBSD TCP stack.**

I. Introduction

The FreeBSD kernel has recently been enhanced with a modular congestion control (modCC) Kernel Programming Interface (KPI) [1]. The KPI allows congestion control algorithms to be implemented as loadable kernel modules based on FreeBSD's kld[2] facility, which can then be dynamically added and removed at runtime.

The ongoing research and development centered around congestion control, coupled with increasingly diverse network path characteristics and application requirements motivate the need for this work. Researchers and developers can quickly and easily test ideas or develop completely new algorithms in a real operating system network stack. Application developers and system administrators can also use the feature to hone the behaviour and performance of TCP connections or the stack as a whole for their specific requirements. The nett result is that the facility allows FreeBSD's TCP implementation to maintain stability, offer improved user/developer friendliness and remain at the forefront of advancements in this area.

The purpose of this report is to investigate the impact of adding this facility to the kernel. We compare the performance of the TCP stack using the default NewReno congestion control algorithm between the 215163 (pre-modCC) and 217806 (post-modCC) Subversion repository revisions of FreeBSD's "head" (9.x) development branch.

The report proceeds with section II which summarises the changes in the TCP stack between the pre- and post-modCC test kernels, section III describes the test setup, section IV presents the results of the comparative study, and section V concludes.

II. Changes in TCP between revisions 215163 and 217806

We use FreeBSD revision 215163 as the test kernel for pre-modCC and revision 217806 as the test kernel for post-modCC. The main changes to the TCP stack between these revisions are:

1) Adding the modCC hooks into the TCP stack
2) Extracting NewReno from the existing stack into modular form
3) Adding two helper hook [3] points into the TCP stack
4) Increasing the scope of the "fast path" in tcp_input.c

The KPI was implemented by adding function pointer hooks into the TCP stack at key locations which algorithms can hook to manipulate congestion control related state as required. The `ack_received()` hook is called on the receipt of every acknowledgement (ack), and therefore introduces the overhead of an indirect function call for every received ack. It is expected that this would have a small, negative impact on performance. The remaining hooks used by the NewReno module (`after_idle()`, `cong_signal()` and `post_recovery()`) are called relatively infrequently and are not expected to add measurable negative impact compared to the pre-modCC stack.

The two helper hook points were added to the TCP input and output paths respectively, but are only called if

a Khelp [4] module has hooked them (currently only applicable when delay-based congestion control algorithm modules are loaded). In the default case of NewReno being the only module available, the hook points only add the overhead of an `if` statement evaluation to the stack and would therefore have no measurable impact on performance.

Code segment 1 Pre-modCC fast path logic test

```
if (SEQ_GT(th->th_ack, tp->snd_una) &&
    SEQ_LEQ(th->th_ack, tp->snd_max) &&
    tp->snd_cwnd >= tp->snd_wnd &&
    ((!V_tcp_do_newreno &&
      !(tp->t_flags & TF_SACK_PERMIT) &&
      tp->t_dupacks < tcprexmtthresh) ||
     ((V_tcp_do_newreno ||
       (tp->t_flags & TF_SACK_PERMIT)) &&
      !IN_FASTRECOVERY(tp) &&
      (to.to_flags & TOF_SACK) == 0 &&
      TAILQ_EMPTY(&tp->snd_holes)))) {
```

Code segment 2 Post-modCC fast path logic test

```
if (SEQ_GT(th->th_ack, tp->snd_una) &&
    SEQ_LEQ(th->th_ack, tp->snd_max) &&
    !IN_RECOVERY(tp->t_flags) &&
    (to.to_flags & TOF_SACK) == 0 &&
    TAILQ_EMPTY(&tp->snd_holes)) {
```

The "fast path" decision logic pre- and post-modCC is shown in code segments 1 and 2 respectively. The modCC patch removed the `V_tcp_do_newreno` variable and implicitly defaulted it to on (no functional change as it was already explicitly on by default), which removes the midsection logic completely. However the key change was the removal of the `tp->snd_cwnd >= tp->snd_wnd` condition, which was disallowing a significant proportion of regular acks through the fast path. The removal thus slightly improves the performance of ack processing and is expected to have a small, positive impact on performance.

III. Methodology

The experimental testbed is shown in Figure 1. A single TCP source sent through a Dummynet [5] router to a TCP receiver (sink). TCP traffic was generated using Netperf [6]. The TCP source and sink were connected to the Dummynet router with 1 Gbps Ethernet.

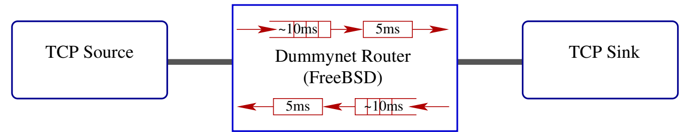

Fig. 1. Experimental Testbed

Pre modular CC	Post modular CC
`tcp_do_segment` (d)	`tcp_do_segment` (d)
	`newreno_ack_received` (a)
	`newreno_cong_signal` (c)
	`newreno_post_recovery` (r)

TABLE I
Kernel functions of interest, Pre and Post modCC

A. Dummynet network emulation

Dummynet was set up to model a 5 ms propagation delay in each direction. To provide TCP with enough network buffering, the Dummynet queue size was set in slots to be approximately 10 ms long, so that the network queue size is roughly equal to the base RTT.

Dummynet operated at a kernel tick rate of 20kHz, giving it a time resolution of 50 μs. Dummynet pipes were set to $P = \{15, 20, 25, 30, 40, 50, 63, 80, 100, 130, 160\}$ Mbps to emulate a network bottleneck link capacity of these values.

B. Load measurements

Measurements of the kernel load were made with `pmcstat` [7], which uses the facilities provided by `hwpmc` [8]. Both test kernels were compiled with the necessary kernel hooks to facilitate the measurements. Using `pmcstat` would semi-regularly panic the kernel due to a spin lock being held to long. To work around this problem, we used the IPMI-based hardware watchdog facility of our test clients and wrote our experimental scripts to redo any trials which were interrupted by a kernel panic.

`Pmcstat` gives a sampling-based statistical estimate of the time spent running different functions in the kernel over a measurement period. Table I lists the functions of interest for the pre- and post-modCC kernels. The modCC framework offloads different aspects of the congestion control process onto dedicated functions in addition to `tcp_do_segment()`. Comparing pre- against post-modCC therefore requires the comparison of cumulative time spent in all functions of interest.

The CPU_CLK_UNHALTED.THREAD_P system mode hwpmc event specifier was used, which samples the CPU's instruction pointer every 65 536 events by default. This event rate was found to produce a sampling rate that was too low to give usable results at the lower bit rates tested. To cater for this we used rates of 24576, 32768, 40960 and 49152 events per sample for the Dummynet pipe rates of 15, 20, 25 and 30 Mbps respectively.

At each Dummynet pipe bottleneck bandwidth setting, measurements were made of the load of the functions of interest. Measurements start 10 s after the TCP source starts sending and continue for a further 1003 s.

C. Congestion window measurements

Measurements of the congestion window evolution were made using the Statistical Information for TCP Research (SIFTR) kernel module [9].

IV. Results

The following results are presented:

- The proportion, p, of hwpmc events that sample the functions of interest versus the achieved throughput (see equation (1)).

$$p = \begin{cases} \frac{d}{T} & \text{Pre-modCC} \\ \frac{d+a+c+r}{T} & \text{Post-modCC} \end{cases} \quad (1)$$

  where T is the total number of hwpmc sampling events and $d, a, c,$ and r are the number of sampling events that occur within the functions of interest from Table I.
- A measure of the work done by the kernels to transmit 1 GB of data (see equation (2)).

$$w = \frac{p}{g} \quad (2)$$

  where p is defined in equation (1), and g is the total amount of data received (in GB) during the test.
- Sample congestion window versus time plots for both pre- and post-modCC kernels.

The achieved throughput results are obtained from Netperf, which reports layer 4 throughput. The Dummynet bottleneck bandwidth is specified as a layer 3 bit rate and therefore the reported throughput is generally slightly less than the configured Dummynet bottleneck bit rate because of protocol overheads.

Where appropriate, graphs show the 20th, 50th, and 80th percentiles (marker at the median, and error bars spanning the 20th to 80th percentiles). Pre-modCC tests were conducted using FreeBSD revision 215163, and post-modCC with revision 217806.

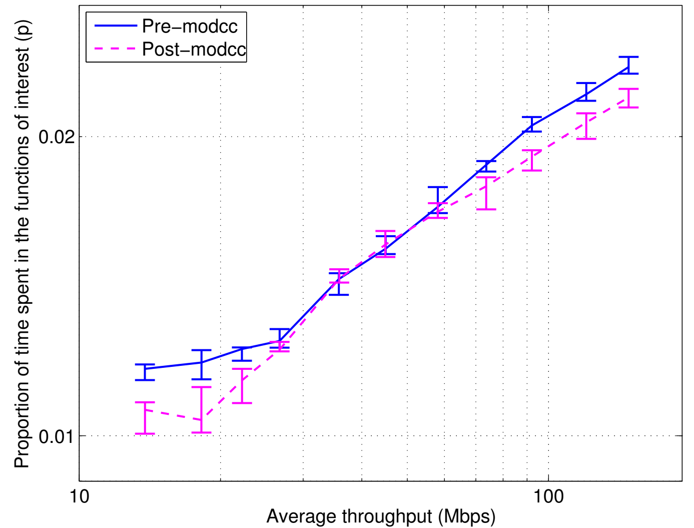

Fig. 2. Plot of proportion of time (as measured by sampled hwpmc events), p, versus achieved bit rate for the different bottleneck rates.

A. Analysis

Figure 2 shows a plot of the median proportion of time (as measured by sampled hwpmc events) spent in the functions of interest during the test. Note that the load increases almost linearly with the achieved throughput. There is little absolute difference between the total function loads pre- and post-modCC. At high tested loads, the post-modCC kernel performs slightly better than the pre-modCC kernel. At the higher bottleneck rates, tcp_do_segment() is mainly processing simple acknowledgements, with longer times between congestion events. The "fast path" modification is of most benefit here, and is evident in the relative improvement in the performance of the post-modCC kernel.

At low loads, there is also a significant relative difference between the pre- and post-modCC kernels. The "fast path" modification does not provide significant benefits, since the congestion window (cwnd) is smaller, and congestion events are frequent. Investigating why the difference exists uncovered a statistically significant difference in the number of sampled events within the tcp_sack_doack() function between the pre- and post-modCC test results. The sample size is reasonable, the results are consistent, the testbed and experiments are identical for both cases.

This indicates that the pre- and post-modCC kernels experience a different number of congestion events which would partially explain the results. As there are also no functional changes to tcp_sack_doack() between the tested FreeBSD revisions, we suspect the

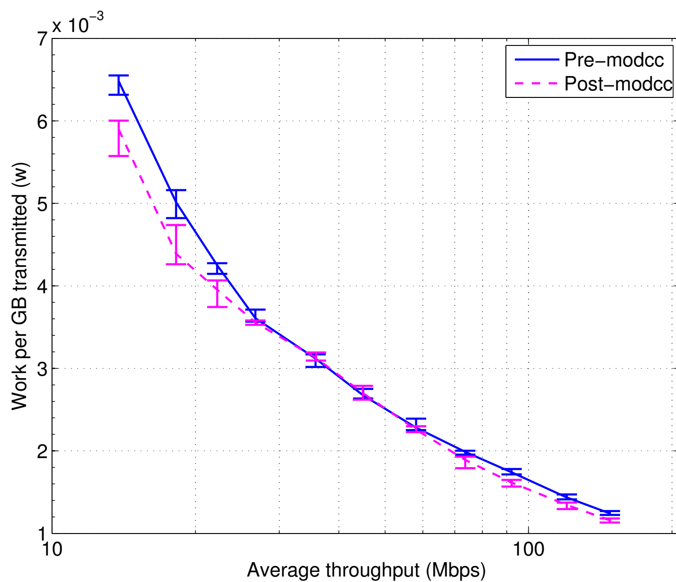

Fig. 3. Plot of the kernel work, w, versus the achieved bit rate to transmit 1 GB of data for the Pre and Post modCC kernels.

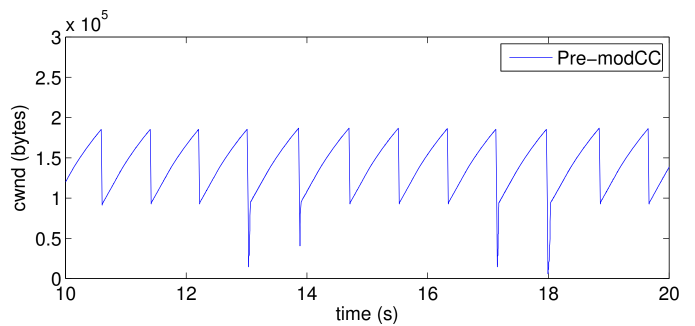

Fig. 4. Plot of cwnd operation for a bottleneck bit rate of 100 Mbps pre-modCC.

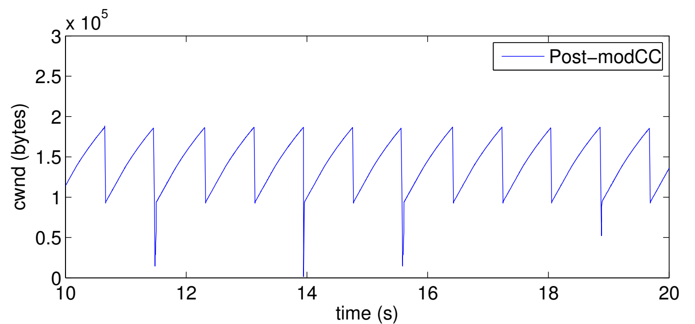

Fig. 5. Plot of cwnd operation for a bottleneck bit rate of 100 Mbps post-modCC.

shape of the post-modCC sub-30Mbps plot data might be an artifact of the `hwpmc` event cycles. Further investigation is beyond the scope of this report.

To give an indication of the relative kernel's work per GB of data transmitted, Figure 3 plots w (defined in equation (2)) versus the achieved throughput for the tested bottleneck link rates. The kernel work drops as the throughput increases. Cwnd oscillates more quickly at lower bottleneck rates than at higher rates, since a smaller number of RTTs are required for cwnd to increase to the point where packet loss occurs. At lower bottleneck rates the more frequent congestion episodes cause the kernel to work harder per unit of data transmitted.

Figures 4 and 5 show that the evolution of cwnd over the course of a connection is the same pre- and post-modCC. They show that the cwnd evolution has not changed at all with the addition of modCC. The occasional downward spikes occur due to the current operation of the TCP stack which causes the connection to slow start after exiting fast recovery under certain circumstances.

V. Conclusion

This study compares FreeBSD 9.x revisions 215163 and 217806 and finds that modifications to the FreeBSD TCP stack which add support for modular congestion control have no negative impact on the performance of TCP in terms of CPU usage. Together with the minor change to `tcp_do_segment()`'s fast path, the changes have a nett small positive impact. Apart from performance considerations, the features add significant functionality to FreeBSD including:

- Allowing system administrators and application developers to choose the most appropriate congestion control mechanism from an available range.
- Making it easier for researchers and developers to implement and test potential improvements to TCP congestion control using a real network stack.

Acknowledgements

This analysis was performed under sponsorship from the FreeBSD Foundation.

References

[1] D. Hayes and L. Stewart, “cc – modular congestion control,” FreeBSD manual 9, Feb. 2011.

[2] D. Rabson, “kld – dynamic kernel linker facility,” FreeBSD manual 4, Nov. 1998.

[3] D. Hayes and L. Stewart, “hhook – helper hook framework,” FreeBSD manual 9, Feb. 2011.

[4] ——, “khelp – kernel helper framework,” FreeBSD manual 9, Feb. 2011.
[5] M. Carbone and L. Rizzo, “Dummynet revisited,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 12–20, April 2010. [Online]. Available: http://doi.acm.org/10.1145/1764873.1764876
[6] R. Jones, “Netperf homepage,” [Accessed 26 April 2010]. [Online]. Available: http://www.netperf.org/
[7] J. Koshy, “pmcstat – performance measurement with performance monitoring hardware,” FreeBSD manual 8, Sep. 2008.
[8] ——, “hwpmc – hardware performance monitoring counter support,” FreeBSD manual 4, Sep. 2008.
[9] L. Stewart and J. Healy, “Characterising the behaviour and performance of SIFTR v1.1.0,” CAIA, Tech. Rep. 070824A, Aug. 2007. [Online]. Available: http://caia.swin.edu.au/reports/070824A/CAIA-TR-070824A.pdf

Appendix

Original Measurements

Originally we attempted to measure the kernel load by looking at the elapsed CPU time given by `ps` for the kernel process. This gave consistent results for experiments conducted at similar times, but varied quite widely for experiments at other times – especially Sunday. We suspect that this may be due to the poor ventilation in the small testbed room, and the fact that the air conditioning system does not run on Sundays. This requires further investigation, but is beyond the scope of this report.