Light-Weight Modular TCP Congestion Control for FreeBSD 7

Lawrence Stewart, James Healy
Centre for Advanced Internet Architectures, Technical Report 071218A
Swinburne University of Technology
Melbourne, Australia
lastewart@swin.edu.au, jhealy@swin.edu.au

Abstract— **With TCP still responsible for the bulk of data transfer over IP networks, increasing research effort is being made to optimise TCP's behaviour for the increasingly diverse range of potential network conditions. TCP's congestion control mechanism is one of the primary areas of focus for TCP research. In order to facilitate this type of TCP research using the FreeBSD operating system, we have developed a light-weight modular congestion control framework for FreeBSD 7. This greatly lowers the time required by congestion control algorithm implementors and researchers to develop concrete implementations of new algorithms and evaluate them. It also helps to encourage congestion control research using an operating system known for its liberal licence, stability and high performance networking stack. The current version of the patch, which this report is based on, is v0.9.1.**

Index Terms— **TCP, Congestion Control, FreeBSD**

I. Introduction

The evolution of deployment technologies and protocols used to deliver Internet connectivity and services has contributed to the increasingly diverse range of paths present in the wider Internet. With TCP still responsible for the bulk of data transfer over IP networks [1], increasing research effort is being made to optimise TCP's behaviour to help it better adapt to this diversity.

Simulation plays an important role in the TCP research and evaluation process, with tools like NS [2] and OMNeT++ [3] being used to quickly prototype and investigate the suitability of ideas. After simulation, the next step is testbed emulation and wider, real-world testing. This step tends to be orders of magnitude more difficult than simulation owing to the plethora of hardware and software platforms that run TCP stacks and the differences in the stack implementations themselves. It is nonetheless extremely important to observe the behaviour of proposed changes "in the wild" to ensure the theory is not usurped by real-world behaviours and limitations. Researchers are therefore faced with the challenge of implementing their work outside of the simulation environment in a real TCP stack.

TCP's congestion control mechanism is one of the primary areas of focus for TCP research. The Linux TCP stack [4] has taken steps in recent years to modularise the code responsible for implementing the congestion control behaviour. This significantly reduces the work required by researchers in order to implement their new congestion control mechanism in a real operating system for the purpose of emulation and wider real-world testing. A simple framework providing hooks into the relevant parts of the TCP stack means researchers can spend their time refining their work rather than coming to terms with the complexity of the TCP/IP stack code, of which only a small portion relates to congestion control.

The widely used FreeBSD UNIX-like operating system provides a mature, stable and customisable platform, suitable for many tasks. Its academic heritage, liberal licence and renowned TCP/IP stack implementation makes it an excellent TCP/IP networking research platform.

As part of CAIA's NewTCP research project [5], a proposed high-speed TCP congestion control algorithm (H-TCP [6]) is being implemented in FreeBSD 7 based solely on the released specification documents. An implementation of the algorithm by the original authors was already available in Linux as a pluggable congestion control module to serve as a baseline for comparison of our "clean-room" implementation.

The only previous work we are aware of implementing something similar to Linux's modular congestion control for FreeBSD is called CC-TCP [7]. It appears to be overly complicated, heavy-weight and not actively maintained. For these reasons we did not consider CC-TCP to be a viable modular congestion control framework on which to base our research.

To simplify the NewTCP implementation effort, we developed a light-weight, modularised congestion control framework for the FreeBSD 7 TCP/IP stack, which is the topic of this report. The framework is modelled on the modular congestion control code found in the FreeBSD 7 SCTP stack. The current version of the patch, which this report is based on, is v0.9.1.

The report is structured as follows: section II discusses how to obtain and use the patch which implements the framework, section III describes the design and architecture of the framework, section IV outlines how to create new congestion control modules using the framework, section V identifies possible further work and section VI finally concludes the discussion.

II. Obtaining and Using the Patch

The ultimate goal is to have this work merged into the FreeBSD source tree. Until such time as this is a reality, the latest version of the patch can presently be obtained from the NewTCP project's website [8].

You must ensure that the FreeBSD source tree is installed on the system before continuing. Either install it from the installation media you used to install FreeBSD or use CVSup [9] to obtain it. Each patch is released against a particular version of the FreeBSD source tree, so it is important you obtain the same set of sources in order to successfully apply the patch file. Check the header in the patch file for specific information about the version of FreeBSD sources the patch was created against. Whilst the current version of the patch is against the FreeBSD 7 source tree, we suspect minimal effort would be required to backport the patch to FreeBSD 6 and possibly even FreeBSD 5. However, the following steps assume you are working on FreeBSD 7.

Download the patch to the local filesystem. To apply the patch, run the command shown in Listing 1 in a shell as root. Replace "path_to_source_tree" with the local filesystem path to the FreeBSD source tree you wish to patch (typically /usr/src) and replace "path_to_patch_file" with the path to the downloaded patch file.

Listing 1 Applying the patch to a source tree

```
/usr/bin/patch -d path_to_source_tree -p0 < path_to_patch_file
```

Once applied, the kernel needs to be recompiled. Run the commands shown in Listing 2 in a shell as root. Replace "path_to_source_tree" with the local filesystem path to the FreeBSD source tree you patched in the previous step. Reboot the system once the kernel has successfully recompiled and been installed. If you experienced any problems, refer to [10] for more detailed information on recompiling the FreeBSD kernel.

Assuming everything has gone well, you now have a modular congestion control capable FreeBSD system. Congratulations! The NewReno congestion control algorithm is set as the default algorithm on system boot. It is now up to you to load in other congestion control algorithms as kernel modules and use them.

Listing 2 Recompiling the FreeBSD kernel

```
cd path_to_source_tree
make buildkernel installkernel
```

III. Design and Architecture

The FreeBSD SCTP [11] implementation set to debut in FreeBSD 7 has support for modular congestion control. This code was used as the model on which our light-weight modular TCP congestion control framework was based.

Table I provides a summary of the kernel source files affected by the framework, quoting file locations relative to the root of the system source tree (typically /usr/src).

A. A note on TCP fast recovery in FreeBSD

FreeBSD considers Reno style fast recovery [12] and SACK [13] to be forms of fast recovery from packet loss. The current FreeBSD implementation enforces that either form of fast recovery is used, with SACK given precedence if available. Current best practices state that fast recovery from packet loss must be used. As such the fast recovery code has been left in place, and hooks have been added to modularise access to congestion control related events outside of fast recovery.

File	State
sys/netinet/tcp_var.h	Modified
sys/netinet/tcp_subr.c	Modified
sys/netinet/tcp_timer.c	Modified
sys/netinet/tcp_input.c	Modified
sys/netinet/tcp_output.c	Modified
sys/netinet/tcp_cc_functions.h	New
sys/netinet/tcp_cc_functions.c	New

TABLE I
FreeBSD kernel source files affected by the framework at a glance

The *net.inet.tcp.newreno* sysctl variable allows an administrator to enable the improvements to Reno congestion control outlined in RFC3782 [14]. These improvements modified aspects of fast recovery and changed the way the congestion window is set on exiting fast recovery. The sysctl variable has been removed in our patch and the fast recovery improvements are now hard coded and always used. Setting the congestion window when exiting fast recovery has been extracted into our modular framework and can be manipulated by loading a new TCP congestion control module.

B. Configuring TCP congestion control

The framework defines two new sysctl variables: *net.inet.tcp.cc.available* and *net.inet.tcp.cc.algorithm*. The *net.inet.tcp.cc.available* variable provides a read-only comma separated list of available congestion control algorithms. The *net.inet.tcp.cc.algorithm* read-write variable is used to query and specify the default congestion control algorithm. This variable can only be set to an algorithm listed in *net.inet.tcp.cc.available*. In the event that an algorithm currently set as the default is deregistered, *net.inet.tcp.cc.available* will be updated accordingly and *net.inet.tcp.cc.algorithm* will automatically be reset to NewReno.

Whilst not mandated, it is expected that any algorithms exposing algorithm-specific configuration options via sysctl will do so under the *net.inet.tcp.cc* sysctl tree, using the algorithm name to group options for each algorithm e.g. if the "nulltcp" congestion control algorithm exposed a variable named "var1" via sysctl, it should use the following sysctl hierarchy to do so: *net.inet.tcp.cc.nulltcp.var1*

C. Implementation Details

The TCP control block struct defined in tcp_var.h has two new members. The "struct tcp_cc_functions *cc_functions" member stores a pointer to the set of functions the connection associated with the control block will use for congestion control. The "void *cc_data" member can be used to attach malloc'd memory to the control block as required. It accomodates the potential need of a congestion control algorithm requiring additional memory per connection to operate. It is the algorithm implementor's responsiblity to manage the pointer. Typically, the memory requisition and release would occur in the algorithm's init() and deinit() functions respectively, but this is not a requirement.

Listing 3 shows the definitions from sys/netinet/tcp_cc_functions.h which are the basic housekeeping functions and global variables required by the framework.

Listing 3 Housekeeping function prototypes and global variable definitions in sys/netinet/tcp_cc_functions.h

```
extern STAILQ_HEAD(tcp_cc_head, tcp_cc_functions) tcp_cc_list;
extern char tcp_cc_algorithm[];
extern struct tcp_cc_functions newreno_cc_functions;
extern int tcprexmtthresh;

SYSCTL_DECL(_net_inet_tcp_cc);
void tcp_cc_init(void);
void tcp_cc_register_algorithm(struct tcp_cc_functions *add_cc);
void tcp_cc_deregister_algorithm(struct tcp_cc_functions *remove_cc);

int newreno_init(struct tcpcb *tp);
void newreno_cwnd_init(struct tcpcb *tp);
void newreno_ack_received(struct tcpcb *tp);
void newreno_post_fr(struct tcpcb *tp, struct tcphdr *th);
void newreno_after_idle(struct tcpcb *tp);
void newreno_after_timeout(struct tcpcb *tp);
void newreno_ssthresh_update(struct tcpcb *tp);
```

The implementation allows multiple algorithms to be available within the kernel at any one time by storing the various tcp_cc_functions structs in a kernel tail queue [15] named "tcp_cc_list".

The name of the default congestion control algorithm is stored in the "tcp_cc_algorithm" string.

The "tcprexmtthresh" variable was originally a statically defined integer in sys/netinet/tcp_input.c. It specifies the number of duplicate acknowledgements required to trigger loss recovery mechanisms. This variable had to be modified slightly to make it accessible from other source files within the framework, requiring it to be extern'd in a header file.

The framework exposes the new *net.inet.tcp.cc* sysctl tree using the "SYSCTL_DECL(_net_inet_tcp_cc)" declaration. The framework exposes two variables under this tree, and allows other congestion control algorithms to hang their sysctl configuration variables from this tree as well.

Given that a vanilla FreeBSD kernel uses NewReno as the congestion control algorithm, the NewReno congestion control module has been hardcoded into the kernel as the default algorithm. This ensures that it is always available to the system for use as a default. The newreno_cc_functions global tcp_cc_functions struct is used where references to a default are required within the TCP stack. The NewReno congestion control related function prototypes are also declared globally so that other congestion control algorithms can call the NewReno functions if they require NewReno behaviour in their algorithms.

The tcp_cc_init() function is called when the network stack is being initialised for the first time during system boot. It initialises the tcp_cc_list tail queue, adds the hard-coded NewReno congestion control module to the tail queue and sets NewReno as the default congestion control algorithm for the system.

The tcp_cc_register_algorithm() function allows a new algorithm to be dynamically registered for use. It takes a pointer to the new algorithm's tcp_cc_functions struct and simply appends the struct pointer to the tcp_cc_list tail queue.

The tcp_cc_deregister_algorithm() function allows a currently registered algorithm to be dynamically removed from the system. It takes a pointer to the existing algorithm's tcp_cc_functions struct. The deregistration process is more involved than that of registration, because current TCP flows could be using the algorithm for their congestion control. The algorithm being deregistered is first removed from the tcp_cc_list tail queue to ensure no new TCP flows use the algorithm. If the algorithm is set as the system default, the default is reset to NewReno. Finally, every TCP control block is checked and any found to be using the algorithm are reset to use the NewReno congestion control module. This final step does not wait for a flow to be in any particular state, so the switch back to NewReno will likely occur while the flow is actively transferring data which will change the flow's dynamics inflight.

Listing 4 shows the definition of the fundamental tcp_cc_functions struct. Each congestion control algorithm is required to define a single instance of this struct.

Algorithms are uniquely identified by their ASCII "name" field, with the maximum length restricted to TCP_CC_MAX_ALGORITHM_NAME_LEN characters (defined in sys/netinet/tcp_var.h).

The "entries" member forms the link used to join tcp_cc_functions structs together in a tail queue. The programmer is not required to maintain this struct member at all.

The remainder of the tcp_cc_functions struct defines function pointers which are called at appropriate places within the TCP stack for algorithm implementors to utilise. Appropriate checks are in place within the TCP stack so that none of the functions must be implemented. This allows the algorithm developer to implement the minimal set of functions required.

Listing 4 Definition of the tcp_cc_functions struct in sys/netinet/tcp_var.h

```
struct tcp_cc_functions {
char name[TCP_CC_MAX_ALGORITHM_NAME_LEN];
int (*init) (struct tcpcb *tp);
void (*deinit) (struct tcpcb *tp);
void (*tcp_cwnd_init) (struct tcpcb *tp);
void (*tcp_ack_received) (struct tcpcb *tp);
void (*tcp_pre_fr) (struct tcpcb *tp);
void (*tcp_post_fr) (struct tcpcb *tp, struct tcphdr *th);
void (*tcp_after_idle) (struct tcpcb *tp);
void (*tcp_after_timeout) (struct tcpcb *tp);
STAILQ_ENTRY(tcp_cc_functions) entries;
};
```

The init() function is called during the initialisation of a TCP control block for a new connection. Any per-flow initialisation required by the congestion control algorithm can be performed here. The function should return 0 on success, or greater than zero on failure. Returning a non-zero value from the init() function will result in the connection being aborted. A pointer to the newly created TCP control block struct is passed into the init() function in case access to the struct's data is required.

The deinit() function is called during the destruction of a TCP control block at the terminiation of a connection. Any per-flow deinitialisation required by the congestion control algorithm can be performed here. A pointer to the TCP control block struct being destroyed is passed into the deinit() function. Releasing malloc'd memory referenced by the TCP control block's cc_data member would typically be performed in this function.

The tcp_cwnd_init() function is called to initialise the congestion window at the very beginning of a connection. If the function is undefined, the initial congestion window is set to the maximum segment size (MSS).

The tcp_ack_received() function is called on the receipt of each TCP acknowledgement, except when in fast recovery mode.

The tcp_pre_fr() function is called on receipt of a third duplicate ACK, prior to entering fast recovery mode. This allows pre fast recovery state to be recorded.

The tcp_post_fr() function is called when a connection has recovered from packet loss and exits fast recovery mode.

The tcp_after_idle() function is called before sending any new data after a period of idleness experienced by a connection.

The tcp_after_timeout() function is called each time the TCP retransmit timer fires.

IV. Developing New Congestion Control Algorithms

With the knowledge gained from the previous section, we can now demonstrate the creation of a very simple congestion control algorithm as a loadable kernel module. You may wish to refer to [16] for a more in depth discussion on programming the FreeBSD kernel.

The congestion control algorithm we are going to create is named NullTCP. It aims to keep the congestion window equal to MSS, which is useful for debugging purposes and not much else. The makefile and source code are included in Appendix A and B respectively. The makefile contents should be placed into a file named "Makefile" and the source code should be placed into a file named "nulltcp.c", both in the same directory. Assuming the FreeBSD 7 source tree exists at /usr/src, simply running "make" on the command line in the directory containg both files should compile and link the module, producing "nulltcp.ko" in the directory.

The noteworthy aspects of the NullTCP implementation are as follows:

- Using a kernel module to implement the algorithm simplifies the development and use of the software.
- The makefile specifies an optional CFLAG to include debugging features in the module at compile time. This provides a very quick way to access debugging information when required, as it simply involves a recompilation and reload of the module.
- sys/netinet/tcp_cc_functions.h is included to gain access to the housekeeping functions and variables defined therein.
- A function prototype is defined for each function pointer we wish to overwrite in the NullTCP tcp_cc_functions struct.
- A tcp_cc_functions struct instance is initialised with the algorithm name and the function pointers we wish to implement. For demonstration purposes, NullTCP implements all of the available function pointers.
- On module load, tcp_cc_register_algorithm() is called and the pointer to nulltcp_cc_functions is passed in. This registers the NullTCP module with the framework and makes it available for use.
- On module unload, tcp_cc_deregister_algorithm() is called and the pointer to nulltcp_cc_functions is passed in. This deregisters the NullTCP module from the framework and makes it unavailable for further use.

Note that had we not wanted to implement all of the function pointers in the NullTCP tcp_cc_functions struct, we could have explicitly set the pointers equal to NULL rather than implementing an empty function.

V. Further Work

Further verification and testing of the framework is required in order to be sure the implementation has not adversely affected FreeBSD's TCP stack in any noticeable way compared to a vanilla kernel. The NewReno congestion control algorithm, now adapted for use in the framework, also needs to be tested against a vanilla FreeBSD 7 kernel to ensure the algorithm is behaving as it did previously.

This additional verification and testing of the framework will lead towards getting the patch integrated into the mainline FreeBSD source tree. A consultation process with the FreeBSD kernel development team will be undertaken in order to work towards this goal.

A nice-to-have feature that should probably be added to the framework is the ability to override the system default congestion control algorithm using a socket option at runtime. The Linux modular congestion control framework offers this possibility, which is particularly useful for simplifying testing procedures with software that supports use of the option e.g. Iperf with the congestion control algorithm selection patch [17].

Using the framework, it would also be useful to begin implementing some of the other TCP congestion control algorithms that exist.

VI. Conclusion

Research into TCP congestion control is ongoing, and requires researchers to experiment with real networking hardware and software in addition to the more abstract algorithms themselves.

We have implemented a light-weight modular congestion control framework for the FreeBSD 7 operating system's TCP stack. Whilst the current version of the patch is against the FreeBSD 7 source tree, we suspect minimal effort would be required to backport the patch to FreeBSD 6 and possibly even FreeBSD 5.

The framework allows congestion control algorithms to be implemented as dynamically loadable kernel modules, which are simpler to develop, debug and distribute. This significantly reduces the amount of work required by congestion control implementors and researchers to evaluate algorithms using FreeBSD. It also makes

FreeBSD a more useful platform for congestion control research than it was previously.

More testing and verification work is required to fully ensure the framework does not noticeably impede or alter the operation of the FreeBSD TCP stack. This is currently ongoing work.

VII. 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

[1] M. Fomenkov, K. Keys, D. Moore, K. Claffy, “Longitudinal study of Internet traffic in 1998-2003,” in *Winter International Symposium on Information and Communication Technologies (WISICT)*, Cancun, Mexico, January 2004. [Online]. Available: http://www.caida.org/publications/papers/2003/nlanr/nlanr_overview.pdf%

[2] “The Network Simulator - ns-2,” Accessed 19 Nov 2007. [Online]. Available: http://www.isi.edu/nsnam/ns/

[3] “OMNeT++ Community Site,” Accessed 19 Nov 2007. [Online]. Available: http://www.omnetpp.org/

[4] I. McDonald, R. Nelson, “Congestion control advancements in Linux,” in *linux.conf.au 2006*, Dunedin, New Zealand, January 2006. [Online]. Available: http://wand.net.nz/~iam4/papers/congestion_lca06_paper.pdf

[5] “The NewTCP Project,” May 2007, Accessed 19 Nov 2007. [Online]. Available: http://caia.swin.edu.au/urp/newtcp

[6] D. Leith, R. Shorten, “H-TCP: TCP for high-speed and long-distance networks,” in *Second International Workshop on Protocols for Fast Long-Distance Networks*, Argonne, Illinois USA, February 2004. [Online]. Available: http://www.hamilton.ie/net/htcp3.pdf

[7] W. Xiuchao, “Congestion Control TCP,” November 2005, Accessed 19 Nov 2007. [Online]. Available: http://www.comp.nus.edu.sg/~wuxiucha/research/reactive/cctcp/index.htm%l

[8] “NewTCP project tools,” May 2007, Accessed 19 Nov 2007. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/tools.html

[9] The FreeBSD Project, “Using CVSup,” Accessed 19 Nov 2007. [Online]. Available: http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/cvsup.html

[10] ——, “Building and Installing a Custom Kernel,” Accessed 19 Nov 2007. [Online]. Available: http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/kernelconfig%-building.html

[11] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang and V. Paxson, “RFC 2960: Stream Control Transmission Protocol,” October 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2960.txt

[12] M. Allman, V. Paxson and W. Stevens, “RFC 2581: TCP Congestion Control,” April 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2581.txt

[13] M. Mathis, J. Mahdavi, S. Floyd and A. Romanow, “RFC 2018: TCP Selective Acknowledgement Options,” October 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2018.txt

[14] “RFC 3782: The NewReno Modification to TCP’s Fast Recovery Algorithm,” April 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3782.txt

[15] FreeBSD Hypertext Man Pages, “QUEUE,” January 1994, Accessed 19 Nov 2007. [Online]. Available: http://www.freebsd.org/cgi/man.cgi?query=queue&sektion=3

[16] L. Stewart, J. Healy, “An Introduction to FreeBSD 6 Kernel Hacking,” CAIA, Tech. Rep. 070622A, July 2007. [Online]. Available: http://caia.swin.edu.au/reports/070717B/CAIA-TR-070717B.pdf

[17] A. Castellani, “Re: Usermode per-flow selection of congestion control algorithm,” March 2006, Accessed 19 Nov 2007. [Online]. Available: http://archive.ncsa.uiuc.edu/lists/iperf-users/mar06/msg00019.html

Appendix A: NullTCP Makefile

```
SRCS=nulltcp.c
KMOD=nulltcp

# Uncomment this define to enable debugging options
CFLAGS+=-g -DNULLTCP_DEBUG

.include <bsd.kmod.mk>
```

Appendix B: NullTCP source code

```
/*
 * Copyright (c) 2007, Centre for Advanced Internet Architectures
 * Swinburne University of Technology, Melbourne, Australia
 * (CRICOS number 00111D).
 *
 * This software was developed by James Healy <jhealy@swin.edu.au>
 * and Lawrence Stewart <lastewart@swin.edu.au>
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. The names of the authors, the "Centre for Advanced Internet Architectures"
 *    and "Swinburne University of Technology" may not be used to endorse
 *    or promote products derived from this software without specific
 *    prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHORS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

//*******************************************************
// nulltcp
//
// A TCP cc algorithm that is designed to keep the cwnd at 1 MSS.
// Useful for debugging purposes, awful in the real world.
//
// Date: November 2007
//*******************************************************


#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/module.h>
#include <sys/socketvar.h>
#include <netinet/tcp_cc_functions.h>
```

```
#define MODNAME "nulltcp - Null TCP congestion control"
#define MODVERSION  "1.0"

int nulltcp_init(struct tcpcb *tp);
void nulltcp_deinit(struct tcpcb *tp);
void nulltcp_cwnd_init(struct tcpcb *tp);
void nulltcp_ack_received(struct tcpcb *tp);
void nulltcp_pre_fr(struct tcpcb *tp);
void nulltcp_post_fr(struct tcpcb *tp, struct tcphdr *th);
void nulltcp_after_idle(struct tcpcb *tp);
void nulltcp_after_timeout(struct tcpcb *tp);

struct tcp_cc_functions nulltcp_cc_functions = {
  .name = "nulltcp",
  .init = nulltcp_init,
  .deinit = nulltcp_deinit,
  .tcp_cwnd_init = nulltcp_cwnd_init,
  .tcp_ack_received = nulltcp_ack_received,
  .tcp_pre_fr = nulltcp_pre_fr,
  .tcp_post_fr = nulltcp_post_fr,
  .tcp_after_idle = nulltcp_after_idle,
  .tcp_after_timeout = nulltcp_after_timeout
};

int
nulltcp_init(struct tcpcb *tp)
{
#ifdef NULLTCP_DEBUG
  printf("initialising tcp connection 0x%x with nulltcp congestion control\n",
(unsigned int)tp);
#endif
  return 0;
}

void
nulltcp_deinit(struct tcpcb *tp)
{
#ifdef NULLTCP_DEBUG
  printf("deinitialising tcp connection 0x%x with nulltcp congestion control\n",
(unsigned int)tp);
#endif
}

void
nulltcp_cwnd_init(struct tcpcb *tp)
{
#ifdef NULLTCP_DEBUG
  printf("Connection 0x%x nulltcp_cwnd_init()\n", (unsigned int)tp);
#endif
  tp->snd_cwnd = tp->t_maxseg;
}

void
nulltcp_ack_received(struct tcpcb *tp)
{
#ifdef NULLTCP_DEBUG
  printf("Connection 0x%x nulltcp_ack_received()\n", (unsigned int)tp);
#endif
}

void
nulltcp_pre_fr(struct tcpcb *tp)
```

```
{
#ifdef NULLTCP_DEBUG
  printf("Connection 0x%x nulltcp_pre_fr()\n", (unsigned int)tp);
#endif
}

void
nulltcp_post_fr(struct tcpcb *tp, struct tcphdr *th)
{
#ifdef NULLTCP_DEBUG
  printf("Connection 0x%x nulltcp_post_fr()\n", (unsigned int)tp);
#endif
  nulltcp_cwnd_init(tp);
}

void
nulltcp_after_idle(struct tcpcb *tp)
{
#ifdef NULLTCP_DEBUG
  printf("Connection 0x%x nulltcp_after_idle()\n", (unsigned int)tp);
#endif
}

void
nulltcp_after_timeout(struct tcpcb *tp)
{
#ifdef NULLTCP_DEBUG
  printf("Connection 0x%x nulltcp_after_timeout()\n", (unsigned int)tp);
#endif
}

static int
init_module(void)
{
  tcp_cc_register_algorithm(&nulltcp_cc_functions);

  uprintf("Loaded: %s v%s\n", MODNAME, MODVERSION);

  return 0;
}

static int
deinit_module(void)
{
  tcp_cc_deregister_algorithm(&nulltcp_cc_functions);

  uprintf("Unloaded: %s v%s\n", MODNAME, MODVERSION);

  return 0;
}

static int
nulltcp_load_handler(module_t mod, int what, void *arg)
{
  switch(what)
  {
    case MOD_LOAD:
      return init_module();
      break;

    case MOD_QUIESCE:
    case MOD_SHUTDOWN:
      return deinit_module();
      break;

    case MOD_UNLOAD:
      return 0;
      break;

    default:
      return EINVAL;
      break;
  }
}

static moduledata_t nulltcp_mod =
{
  "nulltcp",
  nulltcp_load_handler,
  NULL
};

DECLARE_MODULE(nulltcp, nulltcp_mod, SI_SUB_KLD, SI_ORDER_ANY);
```