Congestion control advancements in Linux

Ian McDonald and Dr Richard Nelson
Department of Computer Science
University of Waikato
Hamilton, New Zealand
Email: imcdnzl@gmail.com, richardn@cs.waikato.ac.nz

Abstract

This paper describes the recent advancements in network congestion control in the Linux kernel. Specifically the paper focuses on the TCP congestion framework, and the implementation of the DCCP protocol stack.

Linux has had multiple TCP congestion methods added to it and the subsequent growth of the codebase has made development difficult. As a result a congestion control framework has been introduced.

This paper also outlines how the congestion control framework was used to implement TCP–Nice. TCP–Nice is an experimental congestion control mechanism that uses less than it's fair share of bandwidth when there is congestion, much like nice does for CPU usage by processes in the Unix operating system.

1 Introduction

Congestion became an issue in the 1980's in TCP/IP networks as documented by Nagle [23]. During this period TCP/IP links on the Internet became increasingly congested and Van Jacobsen [13] in 1988 proposed that if "conservation of packets" was observed then TCP flows would be generally stable. The "conservation of packets" was implemented by a congestion window which would be dynamically resized until the connection reached an initial state of stability, and as conditions changed. More packets would not be added to the congestion window when it was full until another was removed after receiving an acknowledgment. These changes are widely credited with preventing ongoing TCP collapse.

At present there is an effort to abstract much of the TCP codebase and move it into a generic IP implementation. The reasoning behind this is to facilitate the implementation of new protocols like DCCP and to improve the implementation of existing protocols such as SCTP.

The rationale behind Datagram Congestion Control Protocol (DCCP) and the current status of the protocol is also discussed. DCCP is a new IP based transport protocol which aims to replace TCP and UDP in some uses. The implementation of the DCCP protocol in the Linux kernel is outlined.

2 TCP Congestion

TCP congestion control, as described by Jacobsen, works on a congestion window system which allows a window of unacknowledged packets. This is initially this is set to one packet and is then increased by one packet per acknowledgment which has the effect of almost doubling the window size per round trip time

(RTT). This continues until either:

- the maximum window size is reached
- the slow start threshold is reached
- congestion occurs.

When the slow start threshold is exceeded the congestion window then increases by a maximum of one packet per RTT. If congestion occurs (detected through multiple duplicate ACKs or timeout) then the congestion window and slow start threshold are altered. A more detailed explanation is provided by Stevens [28] and RFC 2581 [1].

This section talks about the more recent TCP congestion changes to Linux. Sarolahti and Kuznetsov [26] describe earlier TCP congestion implementation in Linux. Linux 2.4.x had TCP New Reno implemented by Alexey Dobriyan.

TCP researchers were working on the Net100 project for Linux TCP performance which became the Web100 project [30]. Stephen Hemminger merged parts of this code that meet the needs of the community and had suitable licenses.

The current list of TCP implementations in the Linux kernel are:

- Reno is the implementation of Van Jacobson's research [13] and was the default congestion control scheme until recently.
- Binary Increase Congestion Control (BIC) [31] was implemented into the kernel in 2.6.6 and the default changed from Reno to BIC in 2.6.8 after Stephen Hemminger investigated data from Standford Linear accelerator tests [27]. The initial implementation of BIC has issues which are described in Li and Leith [16]. This has been resolved in the 2.6.11 kernel. BIC aims to address issues on high performance networks, particularly around RTT unfairness, and uses a combination of additive increase and binary search to alter the size of the congestion window.
- Vegas [2] is based on Reno and tries to track the sending rate through looking at variances to the RTT along with other enhancements.
- Westwood [18] is an implementation that estimates the available bandwidth and is claimed to be suited to wireless use or other networks where loss may occur which does not mean congestion.
- TCP–Hybla [3] is a congestion control mechanism that works with links such as satellite which have high RTT but also high bandwidth as some other congestion control mechanism's favour low RTT flows.
- H–TCP [7], Highspeed TCP [4], and Scalable TCP [14] all aim to improve congestion control on high speed networks.

As the number of implementations in the Linux kernel increased the code became more complicated and there was not a consistent way to use each congestion control implementation. Stephen Hemminger has rewritten the TCP code to make it more modular. This has involved splitting each algorithm into a separate file in `net/ipv4` and implementing a structure to register these in `include/net/tcp.h` — details are shown in Appendix A.

An implementation which demonstrates this simply is scalable TCP which is implemented in `net/ipv4/tcp_scalable.c`

This new modular structure was implemented in Linux 2.6.13. The latest stable release of 2.6.13 should be used as a minimum though as there were some critical bugs found in the implementation.

As part of this implementation the default TCP congestion control mechanism can be altered

dynamically, either by a sysctl or on a per socket basis by using TCP options. This allows the use of different algorithms for different link characteristics if desired or to allow different applications to use different algorithms.

It is of some concern that the BIC became the default Linux TCP implementation while there will still issues with it and there have been other regressions within the networking stack of Linux at other times also. A testing framework is necessary for the networking stack for each release to be tested against. Stephen Hemminger has carried out some preliminary work in this area [10].

3 TCP–Nice

Much of the focus on TCP congestion control research has been on transmitting the most data possible while maintaining fairness with other data flows and maintaining stability.

At the University of Waikato we are investigating using a TCP congestion control variant which uses less than it's fair share when faced with congestion. The reason for this is to allow large file transfers that are not time critical to occur at a lower priority and allow available bandwidth to be used for other applications.

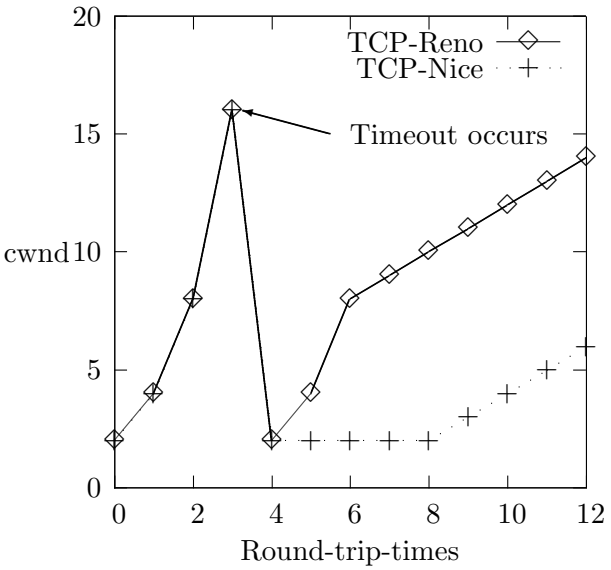
It could be argued that the appropriate place to do this on is a middle box, in a similar manner to the QBone Scavenger Service [25], but often users do not have access to this or the equipment is not capable of this – for example domestic ADSL/cable routers.

The name TCP–Nice is drawn from the use of the nice command in Unix/Linux which lowers a process' priority.

The following characteristics were desired:

- Use less than fair share when congestion loss occurs
- Use less than fair share when competing flows cause congestion
- Make use of available network capacity when no competing traffic causing congestion

Figure 1: cwnd for TCP Nice vs Reno



TCP–Nice would behave similarly to Reno during the startup but be more conservative after a congestion event as shown in Figure 1 where cwnd is the congestion window size.

3.1 Implementation

In this section the new TCP congestion framework for Linux is demonstrated using the implementation of TCP–Nice as an example. The complete code for TCP–Nice is in Appendix B.

To initialise a new TCP congestion control mechanism the `tcp_congestion_ops` structure must be initialised and then a call to `tcp_register_congestion_control` is made. In this case it can be seen that `start`,

rtt_sample, undo_cwnd and get_info are not implemented. It is also possible to use other congestion control functions here – for example Scalable TCP sets min_cwnd to tcp_reno_min_cwnd.

Data can be stored by allocating data referenced through the inet_csk_ca function which points to an area of private data. In TCP–Nice this is used to record the last loss time through the tcp_nice_data structure.

The following code is run when the the TCP congestion state changes:

```
static void tcp_nice_state(struct sock *sk,
u8 new_state)
{
  struct tcp_sock *tp = tcp_sk(sk);
  struct tcp_nice_data *ca =
inet_csk_ca(sk);

  if (new_state == TCP_CA_Loss) {
    tp->snd_ssthresh = 2;
    tp->snd_cwnd = 2;
    ca->last_loss = jiffies;
  }
}
```

In this case the slow start threshold (snd_ssthresh) and congestion window (snd_cwnd) are reduced to two and the time of the loss is recorded.

The congestion avoidance function is implemented as follows:

```
#define LOSS_TIME_CLAMP    4

void tcp_nice_cong_avoid(struct sock *sk,
u32 ack, u32 rtt, u32 in_flight, int flag)
{
  struct tcp_sock *tp = tcp_sk(sk);
  struct tcp_nice_data *ca =
inet_csk_ca(sk);

  if (in_flight < tp->snd_cwnd)
    return;

  if ((jiffies - ca->last_loss) <
(LOSS_TIME_CLAMP * HZ)) {
    tp->snd_ssthresh = 2;
    tp->snd_cwnd = 2;
    return;
  }

  ... existing reno code follows
```

In this case the code is modeled on TCP Reno but keeps snd_ssthresh and snd_cwnd at two if there has been loss in the last LOSS_TIME_CLAMP seconds.

Below are the functions for setting the slow start threshold and minimum congestion window:

```
/* Slow start threshold is quarter the conges-
tion window (min 2) */
u32 tcp_nice_ssthresh(struct tcp_sock *tp)
{
  return max(tp->snd_cwnd >> 2U, 2U);
}

/* Lower bound on congestion window. */
u32 tcp_nice_min_cwnd(struct tcp_sock *tp)
{
  return 2;
}
```

In implementing TCP–Nice it was decided to use a slow start threshold of a quarter the congestion window instead of a half as per Reno and to set min_cwnd to two.

The implementation of TCP–Nice shows that it is relatively simple to implement a new congestion control mechanism in the kernel because of the framework that has been introduced into the Linux kernel.

3.2 Results

TCP–Nice was tested with competing TCP flows to see how the flow was reduced and the time taken to recover. With the code implemented as per the previous section, or slight

variations of this, all of the desired characteristics were not achieved.

To achieve all of the characteristics desired will require further experimentation and/or mathematical modelling.

4 DCCP

Datagram Congestion Control Protocol (DCCP) [15] is a new transport protocol that is at draft RFC status. DCCP is an unreliable session based protocol. This means that it is session based like TCP but unreliable like UDP. The rationale behind the new protocol is that existing protocols do not handle the requirements of modern applications such as multimedia as well as desired. The use of UDP does not provide congestion control at the transport level and is not session based so will not traverse some firewalls or NAT devices. TCP does provide congestion control and a session but is not as suitable due to retransmission and the use of AIMD based congestion response which alters the transmission rate rapidly rather than smoothly. DCCP aims to provide a solution to these problems.

The DCCP protocol is defined in a modular method — there is a base protocol defined while multiple congestion control mechanisms can be implemented through the use of Control IDs (CCIDs). At the time of writing there are two core CCIDs with others proposed:

- CCID2 [5] is TCP–like congestion control and implements a congestion control mechanism based on TCP.
- CCID3 [6] is based on TCP Friendly Rate Control (TFRC) [8]. TFRC aims to provide a smoother response to congestion than TCP while still using a "fair" share of bandwidth compared to other flows. TFRC achieves this goal by estimating the sending rate available rather than halving the window in response to congestion as TCP can do.

4.1 History

For the Linux 2.4.x kernel there were two main early releases of DCCP. There was an implementation by ICIR [11] to test the difficulty of implementing the spec and an implementation by Patrick McManus [20] which implemented the base protocol and CCID2.

Waikato University took the implementation from Patrick McManus and incorporated CCID3 code from Lulea University of Technology [17] relicensed under the GPL. Earlier this year this code was tidied for release and is available for the 2.4.27 kernel [29].

4.2 Implementation

Arnaldo Carvalho de Melo started implementing a version of DCCP for the 2.6.12 kernel. This initially consisted of the base protocol without CCIDs. Parts of the Waikato University code base, which was updated to the 2.6.11.4 kernel, were then merged — mostly in CCID3 support. As part of the code being developed TCP socket code was refactored so that it supported multiple protocols which is discussed later in this paper. This code base has been accepted into the kernel tree by Linus Torvalds and was released in 2.6.14.

One of the challenges of implementing CCID3 was the implementing of the mathematical calculation of the rate. The challenge was two fold — the lack of 64 bit integer operations on 32 bit architecture and the inability to use floating point instructions in the kernel. This was resolved by converting the Lulea floating point lookup tables to an integer based one with some reasonably complex manipulations. As part of this a large amount of testing was carried

out which showed that most implementations to date had implemented the rate calculation incorrectly.

It has proved relatively trivial to port user level applications to DCCP. Netcat, iperf, ttcp, ssh all have had DCCP support added relatively quickly. Programs that depend on the format of the packet e.g. tcpdump, ethereal have taken more effort.

For applications to make full use of the features available in DCCP such as rate and loss feedback a new API will need to be developed. Further research is being carried out in this area by the author [19].

DCCP has been implemented for both IPv4 and IPv6. IP version specific code has been split into ipv4.c and ipv6.c

The directory structure for the DCCP code is:

```
net/dccp             base protocol
net/dccp/ccids       CCID specific
net/dccp/ccids/lib   CCID libraries
```

Further details on the implementing of DCCP can be found in Melo [22].

Tests have been conducted between Brazil and New Zealand by Arnaldo Carvalho de Melo and Ian McDonald using the public Internet which consisted of a route using 18 hops. The tests consisted of using netcat and ttcp which had been modified to use DCCP. Initially the tests failed with checksum failure on the header. This was proven to be due to NAT as the DCCP checksum covers the source and destination IP addresses amongst other fields. With checksum tests temporarily removed the tests proceeded successfully. It is extremely encouraging to see that DCCP was able to traverse the public Internet successfully. From this the conclusion could reasonably be drawn that ISPs are enabling all IP based protocols to travel over their networks.

There also remains work to be done for DCCP to achieve full compliance with the DCCP specification. The two major things that need to occur for this to happen are CCID2 implementation and feature negotiation. Further interoperability testing needs to be carried out once other operating systems implement DCCP.

Devices that implement NAT will have to be modified to allow DCCP to traverse them. The implementation of this will be similar to the implementation of TCP or UDP NAT. To implement NAT, port mapping will need to be put in place and the checksum recalculated as DCCP checksums the pseudo–header in the same way as TCP. This should be a priority to be implemented in future versions of the Linux kernel.

5 IP code restructuring

When Arnaldo Carvalho de Melo started on his implementation of DCCP he realised that there were many similarities between the current TCP code and what would be required for DCCP. The code was restructured so that it could be used for both TCP and DCCP, thus avoiding any replication.

In particular TCP sockets and code using these sockets were examined to see if they could be used in multiple protocols rather than just TCP.

These changes are a continuation of Arnaldo's earlier work [21]. There still remains potential to rework existing protocols such as SCTP to use the restructured code, to extend the code shared between TCP and DCCP, and also to implement other protocols such as XCP.

A similar process has also been commenced by Arnaldo Carvalho de Melo for DCCP over IPv6 as well.

As both TCP and DCCP codebases use modular congestion control mechanisms there is also

scope to consider whether further code can be shared or whether the individual congestion control mechanisms could be used for both protocols.

6 Testing

For congestion control to be effective a variety of network conditions need to be simulated and measured. The following subsections detail programs used in the testing of TCP and DCCP.

6.1 Tcpdump

DCCP support has been added to the development branch of tcpdump which is software used to analyse traffic flows on a packet basis. Tcpdump was used extensively in the testing of DCCP to help resolve bugs. At the time of writing tcpdump is the main packet analysis tool used although there are experimental patches to Ethereal available.

6.2 Netem

Netem [9] is a traffic shaping tool developed by Stephen Hemminger and included in the Linux kernel from 2.6.7. Netem can be used to drop packets, inject delay, duplicate packets and reorder packets. The distribution on the loss, duplication and delay can be uniform or user defined.

Netem only works on the outbound queues so it is necessarily typically to enable it on a PC running as a router with at least two Ethernet cards. An example of testing setup used in TCP and DCCP testing is shown in Figure 2.

Netem draws ideas from Dummynet which is part of FreeBSD and some code from NIST Net [24].

Figure 2: NetEm setup

10.0.2.2/24 10.0.2.5/24 10.0.7.7/24
10.0.7.5/24

6.3 Performance testing

The programs ttcp and iperf [12] were modified to enable use of DCCP and selection of TCP congestion control at run time. These were used to measure throughput speed and loss over a connection.

6.4 Ostra

Arnaldo Carvalho de Melo has developed a tool called Ostra which has proved extremely useful in the development of DCCP and has potential for wider deployment in Linux kernel development. Ostra wraps existing structures within the kernel to capture variables and parameters changing, program flow and timing information. This is implemented in C wrappers being put around existing code. The data is captured at runtime and then Python programs are used to output data including an HTML front end.

7 Conclusion

In recent Linux kernel releases there has been substantial changes made to congestion control.

- BIC has become the standard TCP implementation
- TCP congestion control code has been rewritten in a more modular format

- TCP congestion control can be selected on a per socket basis
- DCCP has been implemented
- TCP code has been refactored in parts of the codebase to support multiple protocols

Linux now has a good base implemented for congestion control mechanisms which gives scope for Linux to be used further for congestion control research. This should enable the networking community to continue to make improvements to IP networking through the use of Linux.

Acknowledgments

We are grateful for those that provided input, reviewed the grammar and critiqued our ideas. Stephen Hemminger of ODSL, Arnaldo Carvalho de Melo of Mandriva and Dr Alan Holt of University of Waikato deserve special mention for their efforts.

A Structure of tcp_congestion_ops

```
struct tcp_congestion_ops {
  struct list_head list;

  /* initialize private data (optional) */
  void (*init)(struct sock *sk);
  /* cleanup private data  (optional) */
  void (*release)(struct sock *sk);

  /* return slow start threshold (required) */
  u32 (*ssthresh)(struct sock *sk);
  /* lower bound for congestion window (optional) */
  u32 (*min_cwnd)(struct sock *sk);
  /* do new cwnd calculation (required) */
  void (*cong_avoid)(struct sock *sk, u32 ack,
    u32 rtt, u32 in_flight, int good_ack);
  /* round trip time sample per acked packet (optional) */
  void (*rtt_sample)(struct sock *sk, u32 usrtt);
  /* call before changing ca_state (optional) */
  void (*set_state)(struct sock *sk, u8 new_state);
  /* call when cwnd event occurs (optional) */
  void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
  /* new value of cwnd after loss (optional) */
  u32  (*undo_cwnd)(struct sock *sk);
  /* hook for packet ack accounting (optional) */
  void (*pkts_acked)(struct sock *sk, u32 num_acked);
  /* get info for inet_diag (optional) */
  void (*get_info)(struct sock *sk, u32 ext, struct sk_buff *skb);

  char name[TCP_CA_NAME_MAX];
  struct module *owner;
};
```

B TCP–Nice code

```
#include <linux/config.h>
#include <linux/module.h>
#include <net/tcp.h>

#define LOSS_TIME_CLAMP    4

struct tcp_nice_data {
    u32    last_loss;
};

void tcp_nice_cong_avoid(struct sock *sk,
u32 ack, u32 rtt, u32 in_flight,
             int flag)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct tcp_nice_data *ca =
inet_csk_ca(sk);

    if (in_flight < tp->snd_cwnd)
        return;
```

```
    if (before(tcp_time_stamp,
ca->last_loss + LOSS_TIME_CLAMP * HZ))
{
        tp->snd_ssthresh = 2;
        tp->snd_cwnd = 2;
        return;
    }
    /* this will keep snd_cwnd and
snd_ssthresh at 2
     * if loss within last x seconds */

        if (tp->snd_cwnd <=
tp->snd_ssthresh) {
                /* In "safe" area, increase.
*/
        if (tp->snd_cwnd <
tp->snd_cwnd_clamp)
            tp->snd_cwnd++;
    } else {
                /* In dangerous area, in-
crease slowly.
         * In theory this is tp->snd_cwnd +=
1 / tp->snd_cwnd
         */
        if (tp->snd_cwnd_cnt >=
tp->snd_cwnd) {
            if (tp->snd_cwnd <
tp->snd_cwnd_clamp)
                tp->snd_cwnd++;
            tp->snd_cwnd_cnt = 0;
        } else
            tp->snd_cwnd_cnt++;
    }
}

/* Slow start threshold is quarter the conges-
tion window (min 2) */
u32 tcp_nice_ssthresh(struct sock *sk)
{
    struct tcp_sock *tp = tcp_sk(sk);

    return max(tp->snd_cwnd >> 2U, 2U);
}

/* Lower bound on congestion window. */
u32 tcp_nice_min_cwnd(struct sock *sk)
{
    return 2;
}

static void tcp_nice_event(struct sock *sk,
enum tcp_ca_event event)
{
    struct tcp_sock *tp = tcp_sk(sk);

    switch(event) {
    case CA_EVENT_FRTO:
        tp->snd_ssthresh = 2;
        break;

    default:
        /* don't care */
        break;
    }
}

static void tcp_nice_state(struct sock *sk,
u8 new_state)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct tcp_nice_data *ca =
inet_csk_ca(sk);

    if (new_state == TCP_CA_Loss) {
        tp->snd_ssthresh = 2;
        tp->snd_cwnd = 2;
        ca->last_loss = jiffies;
    }
}

static struct tcp_congestion_ops tcp_nice =
{
    .ssthresh    = tcp_nice_ssthresh,
    .min_cwnd    = tcp_nice_min_cwnd,
    .cong_avoid    = tcp_nice_cong_avoid,
    .cwnd_event    = tcp_nice_event,
    .set_state    = tcp_nice_state,
    .owner        = THIS_MODULE,
    .name        = "tcp_nice"
};

static int __init tcp_nice_register(void)
{
    return tcp_register_congestion_control(
&tcp_nice );
```

```
}

static void __exit
tcp_nice_unregister(void)
{
    tcp_unregister_congestion_control(
&tcp_nice );
}

module_init(tcp_nice_register);
module_exit(tcp_nice_unregister);

MODULE_AUTHOR("Ian McDonald");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("TCP Nice");
```

References

[1] R. Allman, V. Paxson, and W. Stevens. Tcp congestion, 1999.

[2] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[3] Carlo Caini and Rosario Firrincieli. Tcp hybla: a tcp enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22(5):547–566, August 2004.

[4] S. Floyd. Highspeed tcp for large congestion windows, 2002.

[5] S. Floyd and E. Kohler. Profile for dccp congestion control id 2: Tcp-like congestion control, Accessed 2005.

[6] S. Floyd, E. Kohler, and J. Padhye. Profile for dccp congestion control id 3: Tfrc congestion control, Accessed 2005.

[7] Leith S. Hamilton. H-tcp: Tcp for high-speed and long-distance networks.

[8] M. Handley, S. Floyd, J. Padhye, and J. Widmer. Tcp friendly rate control (tfrc): Protocol specification, January 2003.

[9] S. Hemminger. Network emulation with netem, 2005.

[10] S. Hemminger. Tcp probes, Accessed 2005.

[11] Icir dccp implementation, Accessed 2005.

[12] Nlanr/dast : Iperf - the tcp/udp bandwidth measurement tool, Accessed 2005.

[13] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[14] T. Kelly. Scalable tcp: Improving performance in highspeed wide area networks, 2003.

[15] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (dccp), March 2005.

[16] Yee-Ting Li and Doug Leith. Bictcp implementation in linux kernels. Technical report, Hamilton Institute, February 2004.

[17] Dccp projects, Accessed 2005.

[18] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297, New York, NY, USA, 2001. ACM Press.

[19] I. McDonald. Phd research proposal: Congestion control for real time media applications, 2005.

[20] Dccp implementation by patrick mcmanus, Accessed 2005.

[21] Arnaldo C. Melo. Tcpfying the poor cousins. In *Ottawa Linux Symposium*, July 2004.

[22] Arnaldo C. Melo. Dccp on linux. In *Ottawa Linux Symposium*, pages 305–311, 2005.

[23] John Nagle. Congestion control in ip/tcp internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4):11–17, October 1984.

[24] Nist net home page, Accessed 2005.

[25] Qbone scavenger service (qbss), Accessed 2005.

[26] Pasi Sarolahti and Alexey Kuznetsov. Congestion control in linux tcp. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 49–62, Berkeley, CA, USA, 2002. USENIX Association.

[27] Tcp stacks testbed, Accessed 2005.

[28] Richard W. Stevens. *The Protocols (TCP/IP Illustrated, Volume 1)*. Addison-Wesley Professional, December 1993.

[29] Wand implementation of dccp, Accessed 2005.

[30] The web100 project, Accessed 2005.

[31] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *IEEE Infocom 2004*, 2004.