PROCEEDINGS OF SCIENCE

Implementing DCCP: Differences from TCP and UDP

Andrea Bittau*
University College London
E-mail: a.bittau@cs.ucl.ac.uk

Mark Handley
University College London
E-mail: m.handley@cs.ucl.ac.uk

We describe our experiences in contributing to the implementation of a new protocol, DCCP, in the Linux kernel. Being the first implementation in a main-stream operating system, we are the first ones to explore the implications and the unexpected issues that could arise from developing this protocol. We will focus on how the DCCP implementation differs from that of TCP and the performance issues that we have encountered.

Lighting the Blue Touchpaper for UK e-Science - Closing Conference of ESLEA Project
March 26-28, 2007
Edinburgh

*Speaker.

PoS(ESLEA)003

Component	lines
Ack vectors & feature negotiation.	1,162
Rest of DCCP core.	2,876
Total DCCP core.	4,038
CCID2.	583
CCID3.	1,839
Minimum DCCP (core & CCID2).	4,621
TCP implementation.	8,042
UDP implementation.	1,160

Table 1: Source lines of code for protocol implementations in Linux 2.6.19.

1. Introduction

The *Datagram Congestion Control Protocol* (DCCP) is a transport protocol that does not provide delivery guarantees and has built-in congestion control [3]. One may think of it as UDP with congestion control, or TCP without reliability. This makes DCCP ideal for multimedia applications that prefer, upon packet loss, sending new data instead of old (and now useless) retransmissions. The congestion control algorithm in DCCP is not fixed and applications may choose which one to use by selecting the appropriate *Congestion Control Identifier* (CCID). Currently, there are two CCIDs defined: CCID2 which is TCP-like and CCID3 which is TFRC [2].

In this paper we focus on the implementation issues of DCCP rather than on its design. Our work has been carried out as protocol research in the context of e-VLBI, an application where multiple data streams from different telescopes are correlated to produce an image. The requirements for e-VLBI are transmitting large amounts of data at a (very high) constant bit-rate, and packet loss can be tolerated. TCP is inadequate for e-VLBI due to its bad performance when dealing with large windows. UDP is partially suitable for e-VLBI since it may not be used on shared links (*e.g.* Internet, or shared academic networks) due to its lack of congestion control. DCCP is the best fit—it can transmit data at a constant bit-rate (CCID3) and in the case of congestion, it will back off. In the following sections we will discuss the major implementation differences in Linux between DCCP, TCP and UDP, followed by some performance considerations.

2. Differences from TCP and UDP

Table 1 summarizes the lines of code of different protocols in Linux 2.6.19. DCCP is core-complete, but still missing some optional parts. The code is $\approx 57\%$ of TCP's code size (UDP is much simpler). Part of the reason is that DCCP needs a state machine and mechanisms equivalent to those of TCP in order to be robust against attacks, *e.g.* it needs to detect whether a reset packet is valid (in sequence) before terminating a connection. UDP does not have this complexity and it is generally left to application protocols (if necessary). Complexity in DCCP is added by *ack vectors* and *feature negotiation*. Because DCCP's delivery is unreliable, the protocol may not make use of cumulative acknowledgments as TCP does. Instead, a map representing which packets have been received and which not, much like TCP's SACK, needs to be transmitted and processed (ack

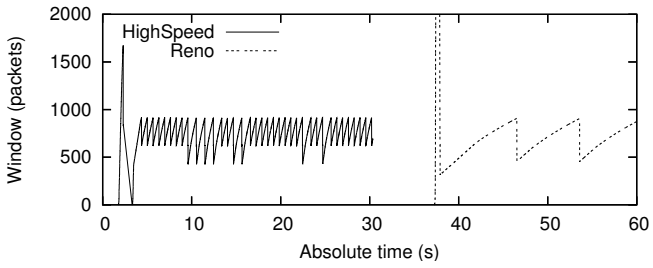

Figure 1: DCCP CCID2 running with TCP's HighSpeed algorithm.

vectors). Thus, detecting loss in DCCP is more complex than in standard TCP. Feature negotiation is a mechanism for negotiating options and may be thought of like TCP options, although the mechanism is much more versatile in DCCP. Together ack vectors and feature negotiation comprise $\approx 29\%$ of the core DCCP code. Because of these extra mechanisms, we believe that the complete DCCP implementation will approach the complexity as TCP's—it is not a trivial protocol like UDP. Currently missing in the implementation are some protocol options (*e.g.* slow receiver) and the handling of special cases such as detecting and dealing correctly with unidirectional data flows.

The CCIDs in DCCP are quite large because they share no code (unlike in TCP's case). This is so because the algorithms are fundamentally different—CCID2 is window based and CCID3 is rate based. We developed an experimental patch which allows TCP congestion control modules to be used by CCID2. Figure 1 shows DCCP's congestion window when using the HighSpeed TCP algorithm [1]. The result is as expected, a higher frequency of losses and a more aggressive window increase when HighSpeed is compared with Reno. It was an interesting result that the same congestion control code worked correctly in protocols which have totally different semantics—reliable *vs.* unreliable. Although DCCP's CCID2 and TCP have very different mechanisms for detecting congestion, the actions taken are very similar. The congestion control modules between TCP and DCCP turned out to be compatible because they only need to be notified about loss—they do not need to detect loss themselves.

3. Performance

We were able to transfer at 1Gb/s, as reported by *iperf*, by using DCCP with CCID2 in a lab experiment. We connected, back-to-back, two Intel Xeon 3GHz boxes with e1000 1Gbit PCIe network cards. On the transmitter, we emulated a 200ms delay using *netem* in order to give us a large bandwidth-delay product (window). This stressed the implementation since the amount of required state, *e.g.* ack vector size, grows proportionally to the window size. We still need to further optimize the code since the CPU utilization is $\approx 90\%$ when transmitting at gigabit rates. In TCP's case, the CPU utilization is lower and this is mainly due to the fact that TCP does not have to process a large ack vector upon receiving every packet.

After profiling the kernel, we discovered that the CPU was spending most of its time calculating checksums ($\approx 25\%$). Checksum offloading to the network card will definitely reduce CPU utilization and we are planning to support it in the future. Figure 2 shows how fast two different boxes can calculate checksums using the Linux kernel code. As the packet size grows, the number

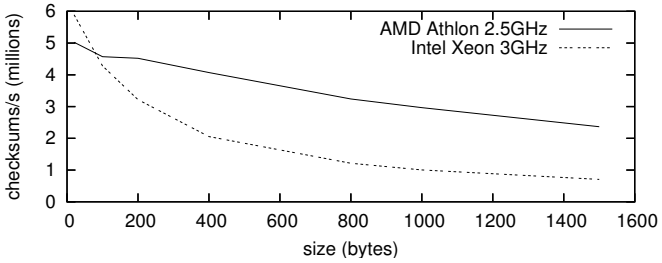

Figure 2: Checksum calculation speed of an (older) Intel and AMD box.

of checksums that can be calculated (thus packets sent) decreases significantly. One difference between DCCP and TCP is that the DCCP protocol allows a sender to specify, via the *checksum coverage* field, which bytes are to be included in the checksum calculation. For example, it is possible to checksum only the header and not the payload but with the drawback of sacrificing some protection. This is tolerable by some applications, such as e-VLBI, and most likely is not an issue in practice if the MAC layer has a checksum too. Thus, by using these simpler to calculate checksums, it is possible to decrease the load on a system. This cannot be done with TCP

In DCCP, packet framing is done by the application so the kernel does not need to worry about segmentation. This leads to a problem in DCCP which is absent in TCP. When sending large chunks of data with TCP, it is possible to invoke a single *send* system call that will cause multiple packets to be sent out. With DCCP, a single *send* call will send out only a single packet. Thus, to transfer large amounts of data, many *send* calls need to be invoked and the context switch overhead is no longer negligible. It is our intent to research APIs which suit DCCP better and are optimized for high-speed networks. For example, we are thinking about a *sendv* system call which will enqueue multiple packets with a single call.

4. Conclusion

The DCCP implementation approaches the complexity of TCP's because of the rich set of features supported by the core protocol. We intend to unify the congestion control algorithms used by TCP and CCID2 in order to share the (at times complex) congestion control code.

In the current implementation, the largest performance hit is checksum calculation. This can be mitigated by offloading checksum calculations to the network card, or in some cases by sending out checksums based only on the packet header and not the entire payload. We also believe that existing APIs need to be extended in order to achieve even greater performance with DCCP, for example, by adding APIs to enqueue multiple packets with a single system call.

References

[1] S. Floyd. HighSpeed TCP for large congestion windows, 2002.

[2] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification, January 2003.

[3] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. In *SIGCOMM '06*, September 2006.