A Congestion-Controlled Unreliable Datagram API

Junwen Lai[*] Eddie Kohler[†]

[*]Princeton University [†]University of California, Los Angeles

lai@cs.princeton.edu, kohler@cs.ucla.edu

We've built an efficient, flexible kernel API for sending unreliable datagrams with DCCP [2], a congestion-controlled protocol. Our API achieves high throughput, kernel-enforced congestion control, and *late data choice*, where the application commits to sending a particular piece of data very late in the sending process. An application using our API can achieve lower-latency transmissions than TCP, since DCCP doesn't retransmit or delay data to enforce ordering constraints, while remaining friendly to the network. Congestion-unaware UDP senders can cause indiscriminate high loss on a slow link, where packets are dropped with the same probability no matter how "important" they are. A DCCP sender using our API can get a much higher fraction of important datagrams, such as MPEG I-frames, through the network, while staying network-friendly.

Applications like streaming and interactive media and online games prefer timeliness to reliability: information has a useful lifetime, after which it's better to send newer information instead. These applications prefer not to use TCP, since the retransmissions needed for reliable bytestream semantics can delay data past its useful lifetime. Unfortunately for the network, the alternative, UDP, is not congestion-controlled except by the application, and TCP-friendly congestion control is tough to implement. In the worst case, congestion-unaware UDP senders could cause congestion collapse.

The Datagram Congestion Control Protocol, or DCCP, is a new protocol that combines TCP-friendly congestion control with unreliable datagram semantics. DCCP presents the user with a clean, UDP-like datagram abstraction. But a sendmsg()-like API wouldn't work well for DCCP, where the kernel might delay sending packets due to congestion. What's needed is a lightweight API that combines kernel-enforced congestion control, high throughput (suggesting a transmission buffer to smooth out burstiness), and late data choice (suggesting application control over the buffer).

Our solution is a *packet ring* data structure, used for send control and synchronization, stored in memory shared between the application and the kernel. The packet ring, like memory-mapped streams [1], resembles the DMA rings used for kernel communication with network devices. The application enqueues packets for transmission by putting them on the end of the packet ring, without bothering the kernel. When the kernel gets control, it can send as many packets as have been enqueued. Keeping several packets on the queue is good for throughput; it smooths out bumps in transmission rates, and allows packets to be rate-paced out even while other applications are running. Unlike memory-mapped streams, applications using packet rings can modify enqueued packets up until the kernel sends them, thus achieving late data choice. See Figure 1.

The shared-memory zero-copy design improves send performance by reducing data copies. Because it reduces control transfers, the packet ring also sends zero-length packets through the kernel faster than conventional UDP. And on a congested network, some packets sent using congestion-controlled DCCP can arrive at the receiver faster than packets sent using constant-bit-rate UDP, and an MPEG-like application using our DCCP API can deliver more than twice as many important "I-frames" than a UDP sender in the same network conditions. See Figure 2.

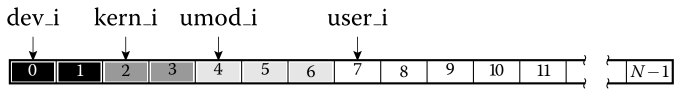

Figure 1—A packet ring. The "X_i" pointers are packet ring indices. The user updates umod_i and user_i, while the kernel updates kern_i and dev_i and checks all four for correctness. The kernel has sent packets 0–1, and 2–3 are ready to be sent. The user previously enqueued packets 4–6 for sending, but is now modifying those packets; they won't be sent until umod_i moves forward.

CBR	**UDP**		**TCP**		**DCCP**	
rate	Loss	Imp't	Loss	Imp't	Loss	Imp't
150 KB/s	62%	10.1%	1.2%	10.0%	0.7%	23.8%
75 KB/s	37%	10.1%	0.5%	10.0%	0.5%	12.8%
60 KB/s	11%	10.1%	0.6%	10.0%	0.6%	10.1%

Figure 2—A CBR application using UDP, TCP, and DCCP over a link limited to 50 KB/s with a token bucket filter. There is some TCP cross traffic. "Loss" measures end-to-end loss rate. "Imp't" is the fraction of packets delivered that were "important": 10% of packets sent were important, but the application would prefer that unimportant packets be dropped when the CBR rate is higher than the network can sustain. DCCP combines a low loss rate with good delivery rate for important packets.

References

[1] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 68–80, Oct. 1991.

[2] E. Kohler, M. Handley, S. Floyd, and J. Padhye. Datagram Congestion Control Protocol (DCCP). Internet-Draft draft-ietf-dccp-spec-05, Internet Engineering Task Force, Oct. 2003. Work in progress.

Based in part upon work done at the ICSI Center for Internet Research and supported by the National Science Foundation under Grant No. 0205519.