Efficiency and late data choice in a user-kernel interface for congestion-controlled datagrams

Junwen Lai[*], Eddie Kohler[†]

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

ABSTRACT

Streaming media servers send many datagrams, very quickly, on many different connections, hopefully without congesting the network. In addition, in order to adapt to constantly changing network conditions, they often demand the capability of *late data choice*, where they choose which data to send immediately before transmission. In this paper, we present a novel user-kernel interface for congestion-controlled datagram protocols that is both *efficient* and *flexible*. Our API consists of a *packet ring* located in shared memory. Applications enqueues packet descriptors directly onto the ring, without crossing the user-kernel boundary, and the kernel dequeues packets from the ring and sends them. This minimizes control and data transfers and gives good throughput. In addition, our API provides a mechanism that allows the application to go back and *safely* change, remove or even reorder previously-enqueued packets, right up until the kernel sends them. This design separates the mechanism for achieving QoS support from any particular QoS policy. We describe the interface and evaluate its performance. By reducing control transfers, the packet ring can even send zero-length packets through the kernel faster than conventional send()-based UDP. Furthermore, on a congested network, an MPEG-like application using our DCCP API can deliver more than twice as many important "I-frames" than a CBR UDP sender in the same network conditions.

Keywords: QoS, media application, DCCP, API, late date choice, congestion control.

1. INTRODUCTION

The factors that limit transfer rates on today's fast networks are often found inside operating systems, and particularly in the interface for transferring data between applications and the network.[1] Much work has been done to optimize this interface by reducing inter-domain control transfers[2, 3] and data transfers,[2–5] eliminating interrupts,[6] and so forth.

Prior work has focused on both TCP performance and, more recently, UDP performance.[7, 8] But emerging congestion-controlled unreliable protocols,[9, 10] such as the Datagram Congestion Control Protocol (DCCP),[11] bring up issues not directly applicable to either TCP or UDP. This kind of protocol was inspired by applications like streaming and interactive media and on-line games, which share a preference for timeliness over reliability. Information has a useful lifetime, after which it is better to drop old information and send newer information instead—an operation that violates TCP's reliable semantics. DCCP senders, then, would like control over the transmit buffer, so that old data can be removed. But unlike UDP APIs, which generally send datagrams immediately, DCCP APIs must implement some transmit buffer to give congestion control power over transmit rates. How can a high-performance API provide control over this transmit buffer? Must it be done with reference to a particular QoS policy,[12, 13] or can it be done generically?

Our goal was to build a fast, flexible, and relatively easy-to-use user-kernel API for DCCP senders (or other congestion-controlled datagram senders) that provided generic control over the transmit buffer, a property we call *late data choice*. We wanted to make it possible for conventional PCs to serve DCCP media to thousands of clients simultaneously. Therefore, we had to avoid control and data transfers and interrupts.

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 the memory-mapped streams abstraction,[3] 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. However, unlike memory-mapped streams, applications using packet rings can safely modify enqueued packets up until the kernel sends them, thus achieving late data choice.

By reducing control transfers, the packet ring can send zero-length packets through the kernel faster than conventional UDP. Furthermore, on a congested network, some packets sent using congestion-controlled DCCP can arrive at the receiver faster than packets sent using simple 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 CBR UDP sender in the same network conditions. This demonstrates both the effectiveness of our API and the possible benefits of congestion control for media applications.

Our main contribution is the packet ring API, and particularly the mechanism that supports late data choice and enables user space data scheduling policy without compromising throughput or latency. It does this by avoiding most, or all, control and data transfers between the sending application and the kernel. Our secondary contribution is the evaluation of this API that demonstrates its low overhead, low latency, and late data choice.

2. RELATED WORK

The Time-lined TCP system[10] for media transfer adds data deadlines to a TCP-like transport protocol. This gives an interesting mix of reliability and unreliability, but forces the application to express all its potential transmission preferences in terms of deadlines. Our application-directed packet ring API is more flexible, allowing the application to implement any policy.

The vast majority of the work on QoS support for multimedia streaming[12, 14, 15] is orthogonal to ours. Our API by itself does not implement, nor is it limited to, any specific QoS policy. Instead, it lets applications implement various QoS or non-QoS policies without sacrificing safety or efficiency. As an example, Section 4 shows how, with our API’s support, an application can adaptively provide media of different levels of quality to clients based on network conditions.

Packet rings are most closely related to Govindan and Anderson’s memory-mapped streams for speeding up continuous media transfer.[3] Like packet rings, memory-mapped streams use shared memory for control and synchronization, hiding the latency of explicit control transfers. However, memory-mapped streams are designed for reliable transfers. There’s no provision for late data choice, and no notification of which packets were delivered. Reliable transfer is a poor match for media applications on possibly-congested networks. The packet ring abstraction adds support for unreliability to memory-mapped streams—specifically, a mechanism by which the application can change or remove packets that have already been queued, and a mechanism by which the application can learn which packets have been delivered.

Using buffer management to reduce the overhead of cross-domain data copies has been a major thread in operating systems research, and variants of this work have been implemented in commodity operating systems like Linux and Microsoft Windows. The basic techniques are virtual memory remapping and shared memory.[16] IO-Lite is the fullest recent development of this idea.[17] These systems pass either descriptors or pages between domains; a descriptor points to the actual data, which is either immutable, copy-on-write, or requires special function calls for an application to access it. None of these systems appreciably reduce cross-domain control transfers; all of them are complementary to our work, which can make good use of any means for reducing data transfers. We chose to implement zero-copy data transfers using special *packet zones* that contain any DCCP packet data. This simplifies our design, but we do not claim it as significantly different from existing work.

A system can remove all cross-domain control and data transfers by transferring data directly between kernel objects, either through a splice() call[2] or current operating systems’ sendfile() APIs. These mechanisms are better suited for reliable transfer, which has far simpler semantics; media transfer works better if more flexibility is given. For example, a media application may dynamically decide which parts of a given stream are less important, and thus can afford to be dropped, on a per-connection basis.

Work in extensible operating systems and user-level networking[18, 19] lets applications access the actual network interface card’s send ring. The DCCP processing that we assume happens inside the kernel would then be taken care of instead by the application, or by a shared library. This would eliminate some small latency introduced by the kernel, and also extend late data choice right up to the moment the network interface grabs a packet to send.

3. PACKET RING

Our interface’s fundamental data structure is the *packet ring*. A packet ring is an array of *packet records*, each of which represents a single packet, coupled with four *indexes* into the array. Two of these indexes divide the packet ring into two regions, packets enqueued for transmission and empty slots. The other two indexes support late packet modification, where the application changes a packet it previously enqueued. These indexes add a third, semantically distinct region to the packet ring: packets that were previously enqueued for transmission, but may now safely be modified by the application.

3.1. Packet Record

The fundamental parts of a packet record identify the packet data to be transmitted using a pointer to the data and a length. The packet ring interface could work with only these three fields, but three more make it more flexible. First, a

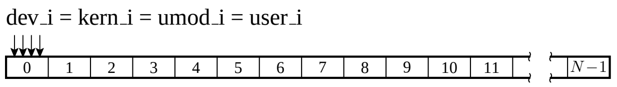

Figure 1. Empty packet ring.

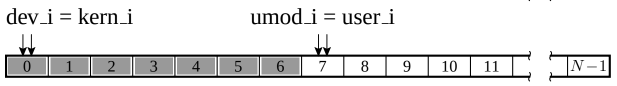

Figure 2. User adds packets, shifting user_i and umod_i. Packets 0–6 are owned by the kernel.

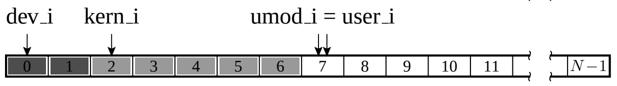

Figure 3. Kernel processes packets 0 and 1 and sends them to the device, shifting kern_i.

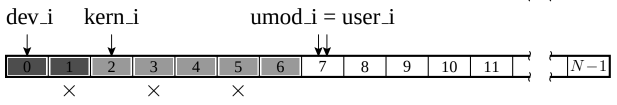

Figure 4. User marks packet 1, 3 and 5 as dead. Note that no index movement is needed and marking packet 1 as dead is safe.

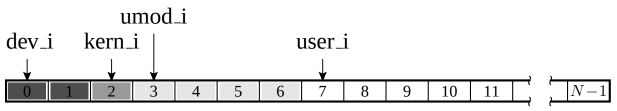

Figure 5. User wants to alter previously-sent packets, moves umod_i. Packets 3–6 are safe to modify.

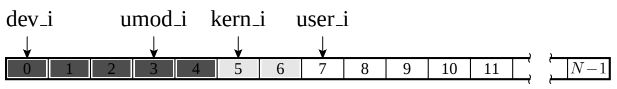

Figure 6. Like Figure 5, but the kernel moved kern_i simultaneously with the user moving umod_i. Only packets 5 and 6 are safe to modify.

user_dead flag lets the application remove a packet from the queue without shifting possibly large numbers of packet records. The kernel will skip over any packets with user_dead flags. Second, a **kern_acked** flag lets the kernel inform the user that a previously-sent packet was received and acknowledged. Finally, the application can assign local sequence numbers to individual packets using a **user_seq** field. The kernel can then inform the application when and if packets with specific user_seq sequence numbers are acknowledged, even if the packet ring has wrapped since the packet was sent (overwriting the kern_acked flag).

3.2. Indexes

Two of the indexes into the packet record array are owned by the kernel, which protects them from accidental modification by keeping private copies. They are:

dev_i, the *device index*, points at the oldest packet enqueued for device transmission. The kernel moves dev_i forward as packets are transmitted by the network device.

kern_i, the *kernel index*, points at the oldest packet that the kernel hasn't processed yet. The kernel moves kern_i forward as it turns packet records into DCCP packets.

The other two indexes are owned by the application. The kernel should keep private copies of these indexes to sanity-check their values, ensuring, for instance, that user_i does not move backwards.

umod_i, the *user modification index*, generally points at the oldest packet record that the user can safely modify. The application moves this index forward to let the kernel transmit new packets, and moves it backward to attempt to modify packets that the kernel hasn't yet transmitted. umod_i is the only index that can move backwards as well as forwards.

user_i, the *user index*, points immediately above the newest packet the application has sent. The application moves this index forward to claim space for packet records it plans to send. It points immediately above the newest packet the application has sent. The application will generally move umod_i and user_i together, unless it wants to go back and modify a previously-sent packet.

When the packet ring is used properly, the kernel and user-modification pointers must lie between the other two pointers: dev_i $\preceq$ kern_i $\preceq$ user_i and dev_i $\preceq$ umod_i $\preceq$ user_i. (Index comparisons are made in circular ring space; we write $x \preceq y$ if and only if $(x - \text{dev\_i}) \bmod N \leq (y - \text{dev\_i}) \bmod N$, where N is the ring size.) The kernel can easily check whether either of these invariants are violated. The ring is empty whenever dev_i $=$ user_i; thus, a ring can hold at most $N - 1$ packets.

These pointers divide the packet records into several contiguous regions.

- dev_i $\preceq i \prec$ kern_i. These packets have been processed by the kernel and are waiting to be transmitted by the network device. The application shouldn't modify the packet records or data. Modifying packet records shouldn't have any effect anyway, while modifying packet data might or might not change the data and/or checksums transmitted.

- kern_i $\preceq i \prec$ umod_i. These packets are available for kernel processing, but haven't been sent to any device. The application shouldn't modify the packet records or data without moving umod_i backwards. Modifying packet records or data might or might not affect the packets that will be transmitted.

•$\max\{\text{kern\_i}, \text{umod\_i}\} \preceq i \prec$ user_i. The application prepared these slots for transmission, but the kernel may not send the enclosed packets yet. This may be because the application moved umod_i backwards in an attempt to modify previously-prepared packets. The max operator handles the case where the application moved umod_i backwards and the kernel moved kern_i forwards simultaneously.

•$i \succeq$ user_i. These slots correspond to previously-sent packets. The kernel may modify some fields in these packet records, to indicate whether the corresponding packets were acknowledged. The application can change packet records or data in this region if it likes, but there's no real point.

Figures 1 through 6 show several examples of packet rings.

3.3. Pacing Transmission

The connection's congestion control mechanism generally determines when packets in the packet ring are actually sent. This is particularly true when the connection's peak rate is limited by congestion—the application provides packets as fast as, or faster than, the connection can consume them. Thus, we always have kern_i $\prec$ umod_i. The kernel will send packets using a combination of TCP-style ack pacing, conventional timers, and rate pacing depending on the congestion control mechanism.

Many connections, however, have application-limited peak rates. That is, the application provides data slower than the connection might consume it, and we often have kern_i $=$ umod_i. To handle this, the kernel could poll application-limited packet rings, sending packets as soon as they become available; but this doesn't scale well as the number of packet rings grows. Alternatively, the application could "wake up" the kernel with an explicit system call every time it adds a packet to the packet ring, or at least when it adds a packet to a previously-empty ring. This would scale better, but it introduces possibly many control transfers.

We split the difference, by adding a packet ring variable called **kern_notify**. The kernel sets a packet ring's kern_notify flag when it decides to stop polling a packet ring for new packets. The application is expected to check kern_notify when it adds a new packet, and make a dccp_notify(packet_ring) system call if kern_notify is true. Thus, the kernel can poll packet rings if it would like, or request explicit notification via dccp_notify() when that would be more prudent. (In our current implementation, the kernel sets kern_notify whenever it runs out of packets to send.) Similarly, the application can enqueue varying numbers of packets before calling dccp_notify().

The more frequently dccp_notify() is called, the more time is wasted in control transfers between application and kernel. We therefore ran experiments where we called dccp_notify() with varying frequencies; the evaluation section presents the results.

3.4. Implementation

The API has been implemented as part of our experimental in-kernel DCCP protocol implementation that runs on Linux 2.4.20. In our prototype, the buffers that store packet data are page-alined, and are allocated from shared memory between user space applications and the kernel that we call *packet zones*. Packets rings and packet zones can be shared amongst multiple threads of a single process or even multiple mutually trustful processes.

For a stored video streaming application, data are loaded from disks directly into a packet buffer using raw disk support or raw file support (In Linux, specify the `O_DIRECT` flag when opening a file) to avoid the copy from file system cache to a packet. For a live audio/video streaming application, the hardware and device drivers also allow data to be directly loaded into a user specified buffer. This implementation decision is not fundamental to our design of the ring API and in fact we plan to extend our API to support other zero-copy transfer mechanisms. However, our design of packet zones does allow us to easily leverage the support in commodity operating systems.

4. EVALUATION

This section quantifies the costs and benefits of the packet ring API. Our performance hypothesis is that the packet ring, combined with congestion-controlled DCCP, can simultaneously achieve low latency, high throughput, and late data choice. We demonstrate low overhead (and therefore high throughput) by comparing with the existing lightweight UDP API. The packet ring can achieve 44% less per-packet overhead than UDP sockets, even in the absence of data copies, by avoiding context switches. We demonstrate late data choice with two sets of experiments. First, on a congested network, we show that the packet ring achieves similar latency to UDP by avoiding retransmission and reordering delay. Second, we perform some experiments on an MPEG-like microbenchmark, showing that our packet ring API (with DCCP) can help an application prioritize its packets, delivering more "important" frames end-to-end than constant-bit-rate UDP or TCP. All the experiments were conducted on machines of identical configurations. These machines use Intel Xeon 2.4

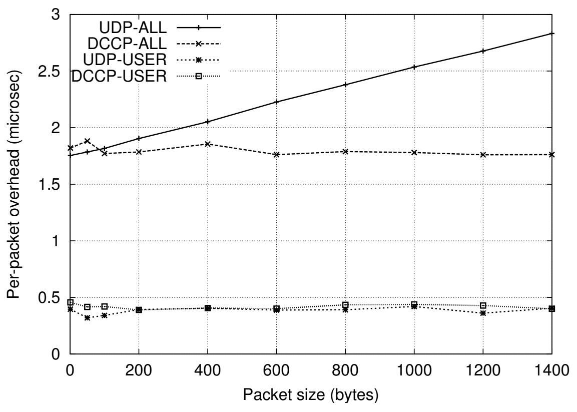

Figure 7. Per-packet overhead for DCCP and UDP senders.

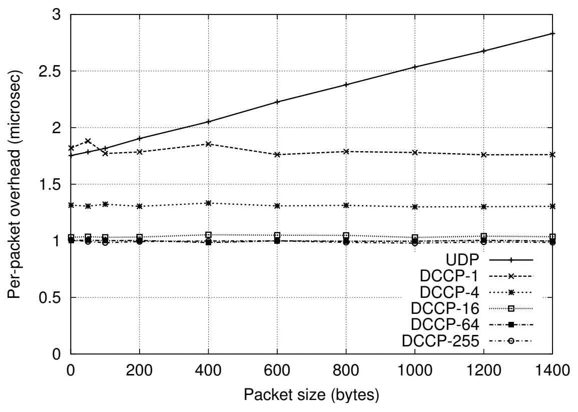

Figure 8. Per-packet overhead for varying context switch frequencies.

GHz processors, 512 MB RAM and Intel PCI EtherExpress Pro 100 Ethernet cards. The operating system is Linux 2.4.20.

4.1. Microbenchmarks

First, we present several microbenchmarks that evaluate the packet ring API's overhead relative to a conventional UDP API which is already very lightweight. We address three questions: How expensive is a packet ring relative to the simpler UDP send()? How effective and how important is the packet ring's elimination of data transfers? And what about its reduction of control transfers? In these experiments, the test packets are sent over a special software Ethernet-card emulator that we have implemented. This measures DCCP protocol and API overhead without polluting the measurements with network effects. The card also pretends to perform hardware DCCP/TCP/UDP/IP checksumming and scatter/gather I/O.

Figure 7 compares the average time cost of sending a single packet using two different APIs (DCCP-ALL and UDP-ALL) and the time spent in user space (DCCP-USER and UDP-USER). The DCCP application always calls dccp_notify() after the packet is placed on the queue. With its zero-copy data transfer capability, DCCP sends small packets as fast as UDP and larger packets much faster. For example, UDP takes 60% more time to send a 1400-byte packet.

Few applications need to call dccp_notify() for every packet, however. In Figure 8, "DCCP-*k*" lines plot the average time cost of sending a packet if dccp_notify() is called only on every *k*th packet. It shows the effectiveness avoiding calling dccp_notify(). For example, DCCP-16 outperforms DCCP-4 by 26% and UDP by 71% or more: even without data copies, context switch overhead can be as high as 44% of overall running time in this microbenchmark. However, reducing the dccp_notify() frequency beyond 1/16 doesn't further reduce per-packet overhead, because user space buffer management time and pure DCCP protocol processing time can't be further compressed. DCCP server applications thus don't need to avoid every possible context switch to achieve optimal performance, which allows them more flexibility in scheduling among different clients.

We also conducted an experiment in which 30 processes open concurrent UDP or DCCP connections and send packets at their maximum rates. The performance trends are almost identical to Figure 8: our API scales well with the number of connections.

4.2. Data Age

In this section, we use end-to-end results from a simple network—two identical machines connected by an emulated point-to-point link—to show that late data choice and DCCP congestion control work, and have advantages for media applications. We use a constant-bit-rate application that sends identically-sized packets at a fixed rate using one of TCP, UDP, and the DCCP API. The packet size equals the TCP MTU (1500 bytes minus packet header length), so that all three protocols act packet-based rather than byte-based. In our experiment, DCCP uses exactly the same congestion control algorithm as TCP. The test flow shares the link with a single, long-lived TCP flow. The link is controlled by a token bucket to reduce its nominal bandwidth to 50 kB/s. Our application generates one packet every 30 ms, which requires more bandwidth than its fair share (roughly 25 kB/s) of the link. The DCCP and TCP senders are limited to their fair share by congestion control, of course, but the UDP sender sends one packet every 30 ms regardless, causing a high loss rate on the link. Finally, the DCCP application leverages DCCP's late data choice feature and marks packets older than three send intervals as dead using the user_dead flag.

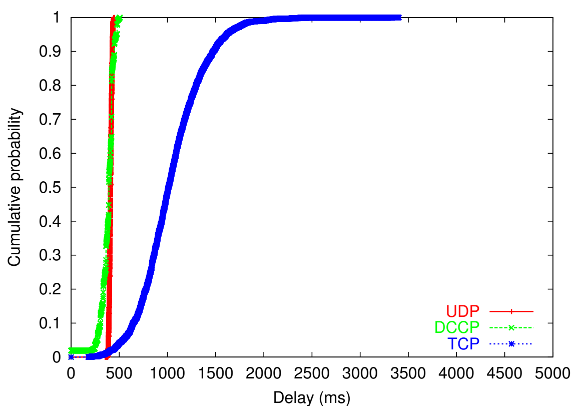

Figure 9. Cumulative distribution of end-to-end delay.

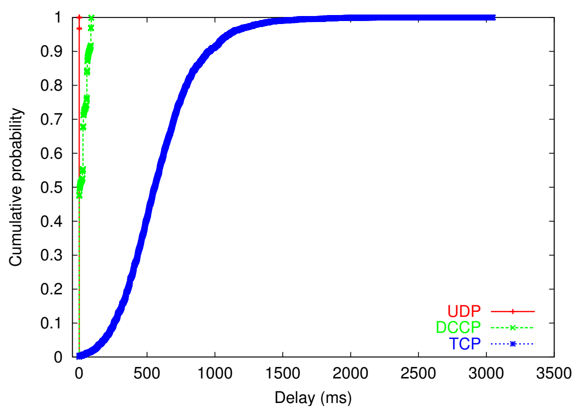

Figure 10. Cumulative distribution of in-sender delay.

Each packet is timestamped at four places: when the packet is sent by the application, when it reaches the sender's device driver, when it gets to the receiver's device driver, and when it arrives at the receiving application. This lets us separate the delays due to different stages of the process.

Figures 9 plots the cumulative distribution function, or CDF, for the end-to-end delay between the sending and receiving applications. DCCP's CDF is very close to UDP's, and both distributions are relatively sharp (most packets have the same delay). TCP's latency distribution varies significantly, however. The average TCP packet is delayed twice as much as the average DCCP packet, even though both flows are congestion controlled and achieve roughly the same bandwidth. This is due to a combination of TCP's reliable transmission, kernel buffering, and in-order delivery. TCP's maximum delay could be lowered somewhat by reducing the kernel's socket buffer size. However, the delay could not be eliminated, and a small socket buffer size would significantly hurt the throughput. Figure 9 also provides a hint of evidence that congestion control might be good for a media flow. About 40% of DCCP packets arrive with lower latency than any UDP packet. This is because the UDP application's fast, constant rate results in a constantly-full link queue.

Figure 10 plots the CDF of the in-sender delay, from when the sender's application sends a packet to when the sender's device driver gets it. In-sender delay captures the delay contribution from reliable retransmissions and kernel buffering. For TCP packets, this component dominates the overall end-to-end delay. For UDP packets, it is negligible since UDP has no kernel buffering or reliable transmission or congestion control. For DCCP packets, this component is also very small due to DCCP's unreliability semantics and its late data choice. TCP application's CDF in Figure 9 has a longer tail than that in Figure 10, the difference comes mostly from TCP receiver side's in-order delivery delay.

4.3. Late Data Choice

We have shown that our packet ring API lets applications avoid sending stale packets by removing stale packets. Our API gives applications much more flexibity than that without bothering the kernel. Consider a simple example. In MPEG encoding, I-frames are key frames while B/P-frames are incremental data. If the total bandwidth requirement to stream a MPEG video is higher than what is available, it is better to drop B/P-frames in favor of I-frames. In our experiment, we assume that 10% of the packets are I-frames, which is in line with some real medium-quality MPEG4 sports videos we collected. All frames have equal size. The link bandwidth is 50 kB/s. The sending application implements a simple late-data-choice algorithm that preferentially removes B/P-frame packets from the packet ring when there are 3 or more packets enqueued; I-frame packets are removed only when removing all stale B/P-frame packets is not enough. We run experiments with sender application sending packets every 10 ms, 20 ms, and 25 ms respectively, all requiring more bandwidth than the link provides.

Table 1 compares the percentages of I-frames among all received frames and the packet drop rates observed at the token bucket link queue using UDP, TCP and DCCP transports. Neither UDP nor TCP is flexible; they treat I-frame packets and B/P-frame packets equally, so roughly 10% of the packets received are I-frames, the same as that in the original encoding. The DCCP application, however, can adaptively change the ratio to according to very different network conditions, which results in only gracefully degraded video quality. Note that this adaptation would be impossible without congestion control. Only after the DCCP sender reduces its rate to avoid congestion losses can it preferentially drop packets, knowing that those remaining will probably get through the network.

The UDP application blindly sends out packets regardless of network conditions, and hence incurs high loss rates. The congestion control mechanisms in TCP and DCCP keep the network loss rate low.

	I-Frame Percentage			Packet Drop Rate		
Send interval	UDP	TCP	DCCP	UDP	TCP	DCCP
10 ms	10.1%	10%	23.8%	62%	1.2%	0.7%
20 ms	10.1%	10%	12.8%	37%	0.5%	0.5%
25 ms	10.1%	10%	10.1%	11%	0.6%	0.6%

Table 1. Comparison of I-frame percentages among all frames received and packet drop rates.

5. CONCLUSION

We have presented the design, implementation and evaluation of an efficient, flexible user-kernel interface for Datagram Congestion Control Protocol. This API uses shared memory to achieve very high throughput; it can reduce the per-packet number of user-kernel data and control transfers to zero. At the same time, it also lets applications flexibly make last-minute choices of what data to send.

Experimental results from our prototype Linux implementation show that applications using our copy-free API outperform comparable UDP applications by up to 60%, and that reducing user-kernel crossings can further improve the performance by up to 71%. The API has low overhead even in the worst case, and scales well with numbers of senders. Most significantly, applications using our API can achieve congestion control, low packet delay and late data choice at the same time: a hypothetical MPEG application using congestion-controlled DCCP sent more than twice as many important "I-frames" as a congestion-unaware UDP application. Thus, a congestion-controlled datagram protocol with a flexible API can form the basis of a responsive streaming media service that adapts to network conditions.

REFERENCES

1. J. Chase, A. Gallatin, and K. Yocum, "End system optimizations for high-speed TCP," *IEEE Communications Magazine* **39**(4), pp. 68–74, 2001.
2. K. R. Fall and J. Pasquale, "Exploiting in-kernel data paths to improve I/O throughput and CPU availability," in *Proc. USENIX Winter 1993 Technical Conference*, pp. 327–334, Jan. 1993.
3. R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," in *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 68–80, Oct. 1991.
4. P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Proc. 14th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 189–202, Dec. 1993.
5. J. Pasquale, E. Anderson, and P. K. Muller, "Container shipping: Operating system support for I/O intensive applications," *IEEE Computer* **27**, pp. 84–93, Mar. 1994.
6. J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. on Computer Systems* **15**, pp. 217–252, Aug. 1997.
7. H. K. J. Chu, "Zero-copy TCP in solaris," in *USENIX Annual Technical Conference*, pp. 253–264, 1996.
8. A. Gallatin, J. Chase, and K. Yocum, "Trapeze/IP: TCP/IP at near-gigabit speeds," in *Proceedings of the USENIX Annual Technical Conference*, 1999.
9. S. Cen, C. Pu, and J. Walpole, "Flow and congestion control for Internet media streaming applications," Tech. Rep. CS-97-03, Oregon Graduate Institute, 1997.
10. B. Mukherjee and T. Brecht, "Time-lined TCP for the TCP-friendly delivery of streaming media," in *Proc. 8th International Conference on Network Protocols*, pp. 165–176, Nov. 2000.
11. 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.
12. J. Huang, C. Krasic, J. Walpole, and W. chi Feng, "Adaptive live video streaming by priority drop," in *EEE Conference on Advanced Video and Signal Based Surveillance*, 2003.
13. S. H. Kang and A. Zakhor, "Packet scheduling algorithm for wireless video streaming," in *12th International Packet Video Workshop*, Apr. 2002.
14. W. Feng, M. Liu, B. Krishnaswami, and A. prabhudev, "A priority-based technique for the best-effort delivery of stored video," in *Proceedigns of SPIE/IS&T Multimedia Computing and Networking*, Jan. 1999.
15. D. Wu, Y. T. Hou, W. Zhu, Y.-Q. Zhang, and J. M. Peha, "Streaming video over the internet: Approaches and directions," in *IEEE Transactions on Circuits and Systems for Video Technology*, Mar. 2001.
16. R. F. Rashid and G. F. Robinson, "Accent: A communication oriented network operating system kernel," in *Proc. 8th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 64–75, Dec. 1981.
17. V. S. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A unified I/O buffering and caching system," *ACM Trans. on Computer Systems* **18**, pp. 37–66, Feb. 2000.
18. R. Bhoedjang, T. Rühl, and H. E. Bal, "Design issues for user-level network interface protocols on Myrinet," *IEEE Computer* **31**, pp. 53–60, Nov. 1998.
19. D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 251–266, Dec. 1995.