Implementation and Evaluation of Late Data Choice for TCP in Linux

Erlend Birkedal[1], Carsten Griwodz[1,2] and Pål Halvorsen[1,2]
[1]Department of Informatics, University of Oslo, Norway
[2]Simula Research Laboratory, Norway
{erlendbi, griff, paalh}@ifi.uio.no

Abstract

Real-time delivery of time-dependent data over the Internet is challenging. UDP has often been used to transport data in a timely manner, but its lack of congestion control is often criticized. This criticism is a reason that the vast majority of applications today use TCP. The downside of this is that TCP has problems with the timely delivery of data. A transport protocol that adds congestion control to an otherwise UDP-like behaviour is DCCP. For this protocol, late data choice (LDC) [8] has been proposed to allow adaptive applications control over data packets up to the actual transmission time. We find, however, that application developers appreciate other TCP features as well, such as its reliability. We have therefore implemented and tested the LDC ideas for TCP. It allows the application to modify or drop packets that have been handed to TCP until they are actually transmitted to the network. This is achieved with a shared packet ring and indexes to hold the current status. Our experiments show that we can send more useful data with LDC than without in a streaming scenario. We can therefore claim that we achieve a better utilization of the throughput, giving us a higher goodput with LDC than without.

1 Introduction

The amount of time-dependent data transmitted over the Internet increases hugely. Many of the applications generating it, like video and audio conferences, on-demand media streaming, multimedia sensor networks and distributed interactive online games, require considerable amounts of data to arrive in a timely manner to provide a satisfactory quality of service. When using the transmission control protocol (TCP) for such applications under network conditions that allow a high bandwidth-delay ratio, this timeliness is a challenge when packets get lost and TCP's generic congestion control mechanisms introduce retransmission delays and reduce bandwidth. Applications may receive data when it has become obsolete in the TCP send buffer. For example, a frame of a layered video may have been displayed already without waiting for the delayed refinement data.

For various reasons, TCP is the most frequently used transport protocol today regardless of application type. It is used both for time-independent and time-dependent applications. Many protocols such as the Microsoft media server protocol and systems like Skype try initially to use UDP, but due to widespread blocking of UDP in firewalls and at ISPs, they have fallback mechanisms to TCP. For example, in logs of the streaming service of the largest online newspaper in Norway, *Verdens Gang*, we found that the initial UDP attempt was successful in only 34.0% of the accesses. The remaining deliveries used MMS over TCP (49.6%) and HTTP over TCP (16.4%) [6]. Many multi-player online games use TCP for the same reasons. Thus, even though we may find appropriate and useful mechanisms for time dependent media delivery in or on top of other transport protocols like UDP, stream control transport protocol (SCTP) [9] and datagram congestion control protocol (DCCP) [7], they are largely ignored by network providers. For this reason we focus on TCP-based solutions.

Using TCP for timely delivery of data poses challenges. Network congestion can result in large packet delays due to the mechanisms used by TCP to achieve reliability and fairness. Packets are frequently held back in the buffer until TCP sends the next segment in the queue. For time-dependent applications, this is particularly bad, because the send buffer contains data for a longer time span. Consider a late video frame in a teleconference or a late position update in a game as illustrations. At a given time, a particular video frame or game position update is to be used for the data presentation at the client, but the sender has not even been able to send the previous data elements. In the next available transmission slot, TCP will then send the next data in the queue, which may be worthless, delaying the new relevant data even further, and possibly out-dating these data as well.

For applications requiring timely delivery of data, the described scenarios waste resources in the network, as delivered data will be less relevant for the receiver. In the current

implementation, TCP does not have any means to deal with the situation. To better utilize the resources and reduce the delay of relevant data, it would therefore be an improvement if an application had mechanisms available to prevent transmission of outdated data, i.e., means to discover and discard network traffic that is already too late. Thus, some control over the transmission buffer is required, enabling modification of buffers that the application has already handed to TCP. Such functionality can be added in several ways. Possible approaches are support for *partial reliability* or *late data choice* (LDC). The partial reliability approach of the stream control transmission protocol (SCTP), for example, is defined as a transport service "that allows the user to specify, on a per message basis, the rules governing how persistent the transport service should be in attempting to send the message to the receiver" [10]. LDC is an alternative implementation for feeding data to TCP that provides a generic control over the transmit buffer [8] without any changes to the TCP protocol itself. It enables an application to control its transmit buffer and gives it the ability to modify or remove data in the buffer.

In this paper, we look at possible ways for delivering time-dependent data faster when delays keep data in the TCP send buffer. To avoid a huge number of cross-platform updates, our aim has been to have a kernel patch which can improve the situation using sender side modification only and which is as closely integrated with the standard Linux TCP implementation as possible without affecting applications that use the normal TCP API. In particular, we present and evaluate an implementation[1] of LDC for TCP in Linux (version 2.6.15.4). Our experimental tests shows that the implementation is able to provide the required functionality while also providing an enhanced data path reducing the number of user-kernel boundary crossings.

The remainder of this paper is organized as follows. In the next section (section 2), we briefly describe the sending operation in Linux TCP before looking at some related work in section 3. Section 4 outlines the main LDC ideas from DCCP, and in section 5, our LDC implementation for TCP in Linux is presented. We evaluate our system in section 6, and a discussion is given in section 7. Finally, section 8 concludes the paper.

2 TCP output

To send data over a TCP connection in Linux, the application uses a system call like `send`, `write`, etc. on a socket with at least a pointer to the data and its size. This call is then managed by the appropriate system call function and sent to the TCP functions for sending data. Here, data is copied from user space into socket buffers in the tail of the socket send buffer. TCP then sends packets from the send buffer in first come first serve order until there is not more data to send (or until it is stopped for various reasons such as congestion or "corking").

The problem for time-dependent data occurs when the transmission has stopped due to lack of space in the TCP send window. The reasons for this are a loss event that is interpreted as a sign for congestion or a flow control event that indicates a blocking receiver. These events force TCP to slow down or stop entirely, respectively. The application may continue to send data to the communication system in spite of this throttling as long as there is space in the send buffer. The data may thus expire in the send buffer if it only has a certain time to live. With no means for the application to notice this situation and react to it, TCP sends the unacknowledged packets in the send buffer in first come first serve order even if newer updated information is placed further back in the queue.

3 Related Work

Even though TCP has some issues with time-dependent data, it is common today to use TCP also for time-dependent streaming and gaming applications. We want to give applications control over the TCP send buffer to ensure that only timely data is sent. In this section, we look at the work most relevant for our work.

The PRTP-ECN extension to TCP [3] makes a tradeoff between reliability and latency by modifying the TCP retransmission scheme. This scheme may acknowledge lost packets at the receiving-side (avoiding a retransmission) and uses ECN to remedy the influence of congestion control. Similarly, TCP Urel[2] is an option at the sender-side to TCP, which sends fresh data in every segment regardless of whether the segment is a new packet or a scheduled retransmission. Furthermore, the SCTP [9] partial reliability extension (PR-SCTP) [10] is able to provide some control over data to be sent and retransmitted. With timed reliability it allows to associate a time to live with a packet. The packet is dropped if transmission is not successful by the deadline, and the receiver advances the cumulative ACK point. Lai and Kohler's LDC API [8] for DCCP [7] allows the application to delete or modify packets sent from the application but not yet transmitted to the network using a shared buffer between the application and the kernel. Finally, dynamic send buffer size [2] adapts the TCP send buffer size in Linux to twice the congestion window size. This allows applications to notice much earlier than the usual implementation when TCP reduces send speed for some reason, and react much faster to it by adapting to a lower bitrate.

[1] Available at http://www.simula.no/departments/networks/software

[2] The paper *TCP Urel: A TCP Option for Unreliable Data Streaming* by Lin Ma, Xiuchao and Wei Tsang Ooi which is still under submission.

All these mechanisms address the challenge of delivering time-dependent data in more or less orthogonal or complementary ways. PRTP-ECN, TCP Urel and PR-STCP (movement of the cumulative ACK) may avoid retransmissions of time-dependent data in case of congestion. PR-STCP (timed reliability), LDC for DCCP and dynamic send buffers enable applications to control what is sent in a congested situation. Of the mechanisms, dynamic send buffers are most similar to LDC, but their reaction speed is considerably lower. What was supposed to be 2 round-trip times worth of data are stored in its send buffers when a congestion event occurs, and it takes subsequently 4 round-trip times to deliver this data, and much of which may become obsolete. Conditionally ignoring lost packets in PR-SCTP and PRTP-ECN require client-side changes, and in the case of PRTP-ECN, the authors rely on ECN, which is not necessarily available. Moreover, TCP Urel (and PRTP-ECN) avoids retransmitting old data independently of the data itself. The DCCP LDC mechanism and the PR-STCP (timed reliability) are in many ways similar, and both can be used in a larger class of applications. As LDC can be used to provide the timed reliability of PR-SCTP, we describe LDC for DCCP in more detail next and then take a look at how to implement LDC support in TCP in the Linux network architecture.

4 Late Data Choice in DCCP

The LDC API [8] for DCCP enhances applications' ability to control, modify and possibly discard the data that has already been sent to the communication system but not been sent to the driver yet. It is implemented as a shared ring between user and kernel space (similar to the ring buffer shown in figure 2) with pointers indicating the current status with respect to the packets that have been sent from the application and that have been processed and transmitted to the network. The former ones may still be controlled by the application. Additionally, the LDC API enables the application to use flags to mark packets for dropping, instructing the kernel to move on to the next packet in the ring buffer. Finally, the kernel can mark packets that have been sent successfully. In this way, LDC lets DCCP retain control over the sending speed and thus, more control over the timely delivery of data to the application.

5 Late Data Choice Support in Linux TCP

This section briefly describes our LDC support for TCP in Linux[3] (for further details, please refer to [1]). The main design is inspired by Lai and Kohler's DCCP API for late data choice (section 4), and the handling of packets that have been sent once is inspired by earlier SCTP proposals. Our primary goal is functionality to be able to go back in the send buffer and delete (or modify) a packet that is still queued in the kernel transmission buffer. We followed also two additional design goals. One was that the implementation should work with an unmodified TCP receiver. The other was to keep the existing TCP stack intact while providing additional LDC support.

[3]We do not address security issues in this paper, but the ring pointers should be protected to avoid illegal updates.

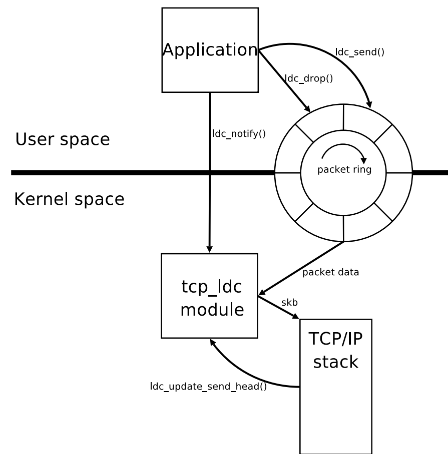


Figure 1. TCP LDC design

As shown in figure 1, depicting the design overview, we use send buffers that are shared between user and kernel space. Furthermore, we have a TCP LDC module in the kernel that links the packet ring and the traditional TCP/IP stack. It enables sending data in the buffer directly using the functions in the original stack. Finally, we have an LDC user space library to provide calls to the LDC functionality. In the current version, it includes functions to initialize the socket as an LDC socket, to send packets using the ring buffer and to drop one or more packets setting the per-packet flags.

The heart of the implementation is the packet ring, which "replaces" the original packet buffer. The main idea is that the application "sends" packets without issuing a system call. It puts data directly into one ring buffer of the packet ring at a time. The kernel transmits data according to TCP's congestion control as long as the packet ring is not empty. If the ring is empty, the LDC kernel module goes to sleep until notified. Thus, when the application has put a packet in the packet ring, it should notify the kernel if it is not already sending packets.

5.1 Indexes and flags

Three index values (or pointers) per connection keep track of status information and are used to manage the ring. They are shown in figure 2 and are used as follows:

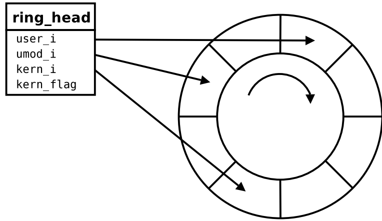


Figure 2. Packet ring and indexes

- The kernel index (kern_i) points to the data element that the kernel should process and send next.
- The user index (user_i) points to the ring buffer that the application should use next.
- The user modification index (umod_i) points to the oldest unsent ring buffer that can be modified safely.

This means that slots with index i where

- kern_i≤i<umod_i contain data ready to be sent
- umod_i≤i<user_i are buffers that the application freely can modify/delete
- user_i≤i<kern_i are free to use

As long as the application does not update data, umod_i equals user_i. The umod_i is only decremented when the application wants to modify or delete a ring buffer. The kernel is then prevented from sending this and all following packet buffers until umod_i is increased again.

Additionally, there are two types of flags. The ring structure contains a kernel_flag. By setting it, the kernel indicates that it is not sending packets and needs to be notified to start sending by setting it. A user_flag exists once for every packet buffer. The application sets it to indicate that the ring buffer has been deleted. The kernel skips it and processes the next buffer in the ring.

5.2 Replacing the existing socket buffer

Adding LDC support requires extra information in the buffer structure and handling of data that differs from the usual write operations. To do this without additional delays, the TCP LDC packet ring mechanism is integrated with the existing TCP socket buffer. With TCP LDC, unsent packets are in the packet ring and can be modified or deleted. Only one packet, the next one to be sent, is in the original TCP send buffer. Ring buffers in the TCP LDC packet ring contain data and a minimal LDC header, while packets in the original TCP send buffer are managed with a traditional Linux skb buffer structure. Ring buffers that are moved to the original buffer are not copied but wrapped as skbs.

5.3 User-Kernel Interface

Our LDC implementation uses a normal TCP socket, and the application manages the kernel LDC functions using setsockopt(), getsockopt() and ioctl() with a set of new options. The socket is first defined as an LDC socket using setsockopt() with the new TCP_LDC option. The tcp_ldc module (see section 5.4) is then loaded on demand. After creating the LDC socket, the application needs access to the allocated memory. This is done using getsockopt() with TCP_LDC. This returns the *user space* address to the shared buffer (packet ring). Now, the application has access to the packet ring and can start placing packets in it. Finally, to wake up a sleeping transmission operation if kern_notify is set, we added an ioctl() request that clears the flag and resumes (or starts) the TCP data transmission.

5.4 The TCP LDC kernel module

The tcp_ldc module is the core of the implementation. It implements buffer initialization and handling, sending packets and removing deleted packets on behalf of the application.

The first step in initializing a socket for LDC is creating a set of pointers. Ring buffers for the packet ring are then allocated in page aligned memory pages and mapped into both the kernel's and the application's virtual memory. After allocating the packet ring, the module "sleeps" until it is notified by an application. Since the packet ring is shared between the application and the kernel, we do not use the normal send (or equivalent) system-calls. Packets are simply copied into ring buffers in shared memory. Since no system call is involved in this, the kernel's send operation can not be initiated in this way. Explicit notification is therefore necessary and achieved by an ioctl.

Sending data does not require a copyin operation like a usual TCP send operation. Instead, tcp_ldc uses the indexes of the ring to find the next data to process and transmit. The data is then sent using the standard skb buffer structure. It copies data with a data pointer to the ring memory just in time for transmission (to avoid unnecessary data movement). The skb is placed on the queue, and the pending frames are pushed out as in normal TCP. All normal TCP operations including retransmissions and congestion control are performed by the originally implemented mechanisms. Furthermore, when TCP has sent one segment, it looks for a new one. For an LDC socket, the traditional socket queue is not used. The TCP transmit function calls the tcp_ldc module to get the next ring buffer element instead. If the ring is empty, the kern_flag is set and the operation goes to sleep.

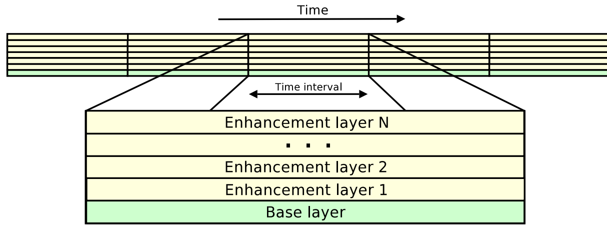


Figure 3. Layered video

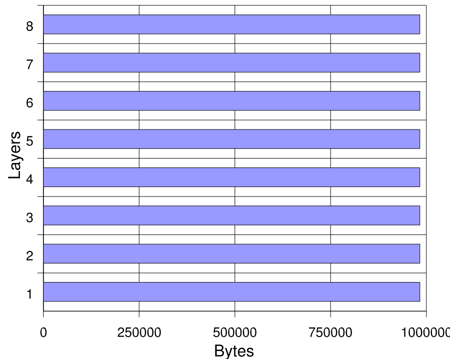


Figure 4. No rate limit, no loss

6 Experiments

We believe that an LDC implementation is useful in scenarios that have very strict time constraints. As a proof of concept, we performed tests to see how LDC could affect the user perception in a video streaming scenario with a minimal latency, as it is desired for video conferencing. Late data in this scenario is consequently dropped at the receiver. We set up an experiment for streaming hierarchically layered video (see figure 3). The server sends with a rate of up to 1 Mbps per stream divided into 8 video quality layers of 128 Kbps. The lowest layer has highest relevance, the other layers each refine all layers below them. Receiving more layers improves the quality of the playout. The application drops packets starting with the higher layers until it can send the stream strictly at playout speed with the first packet as the time reference. We used a test setup with a server and a client connected by one machine running *netem* to emulate a network connecting them.

The tests presented here ran for 60 seconds. The bandwidths 1024 Kbps (full rate), 512 Kbps and 256 Kbps were used. We used a 5% loss rate to provoke delays due to retransmissions. For tests of TCP without LDC support (denoted ”regular TCP”), we used the Linux default (New Reno with DSACK) and the regular `send()` system call. For the LDC tests, we used our modified stack and the LDC user space library to allow the application to send data and drop data using the LDC kernel module.

The measured average values of 10 test runs, using the three stated bandwidths, are shown in figures 5 and 6 for regular TCP and LDC, respectively. In a scenario with unlimited bandwidth and no loss (figure 4), each of the 8 layers should be complete for both TCP variants, i.e., almost 1 MB of data (983040 B) for each layer or about 7.5 MB.

Available BW	TCP	TCP with LDC	Difference
1024 Kbps	862.47	851.96	1.22%
512 Kbps	457.72	456.30	0.31%
256 Kbps	235.88	235.24	0.27%

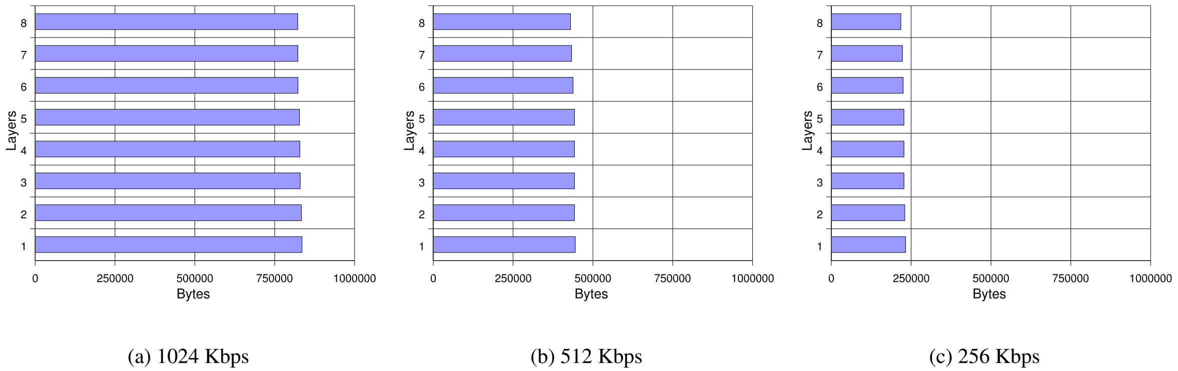
Table 1. Receive rate in Kbps

In the first test, we looked at regular TCP. When we introduced rate limits of 1024 Kbps, 512 Kbps and 256 Kbps (and 5% loss), shown in figures 5(a), 5(b) and 5(c), we received a total of 6.32 MB, 3.35 MB and 1.73 MB of data during the 60 seconds, respectively. As we see in the figures, the received data is evenly distributed (except in the last 1-second, eight layer segment) on the eight layers. The reason for this is that the sending application does not actively drop any packets. Since TCP is a reliable protocol and we send one second of data from one layer at a time starting at the most significant layer (base layer), the application receives all the data from all the layers in the order in which it was sent.

The LDC results with rate limits of 1024 Kbps, 512 Kbps and 256 Kbps (and 5% loss) are shown in figure 6(a), 6(b) and 6(c). We received a total of 6.24 MB, 3.34 MB and 1.72 MB of data in average, respectively. Although we got a little less data in the same time period than with regular TCP (see table 1), we observe that the received data is much more relevant. The base layer, layer 1, is complete for all test scenarios, meaning that we can present the media at the receiver in the correct time-frame. The figures illustrate clearly that we achieved the desired effect, namely that the application was able to prioritize the most significant layers and drop data from the less significant layers when it was not transmitted in time. This effect is a result of dropping packets still in the send buffer after the deadline has expired.

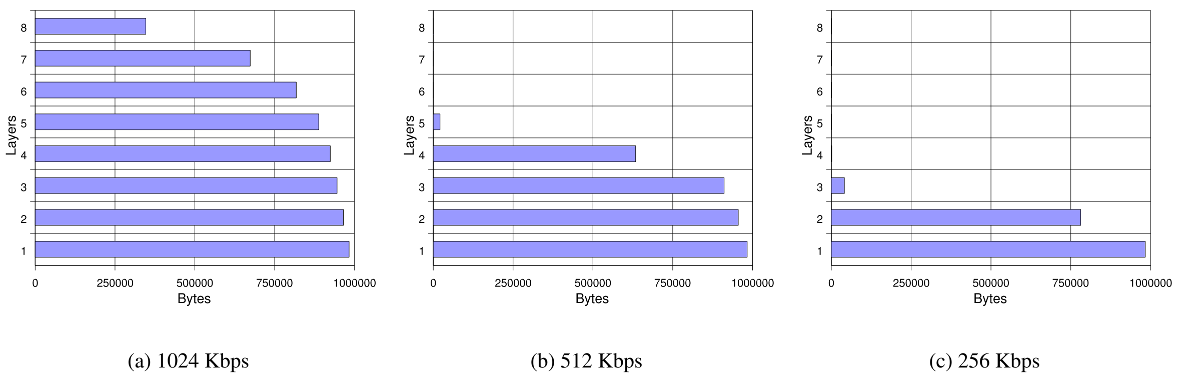
In figure 7, we have looked at the amount of data that arrives in time for a streaming scenario with minimal buffers at the receiver. This scenario is valid for highly time-critical applications but also when streaming to very limited receivers such as mobile phones.

Figure 7(a) shows regular TCP where all data is delivered despite of over-aged packets. In the presence of packet loss, it is unable to deliver data in time for playout. On the other hand, LDC enables the application to distinguish between useful and useless transmissions. This increases the amount of data that can be delivered in time. Similar conclusions can be drawn from figure 8(a) where the arrival time of each packet is plotted according to the time by which the payload should have been played out. LDC delivers data according to the playout consumption, but has dropped some (of the least relevant) packets as shown by the gaps between the points (the other rate limitations are



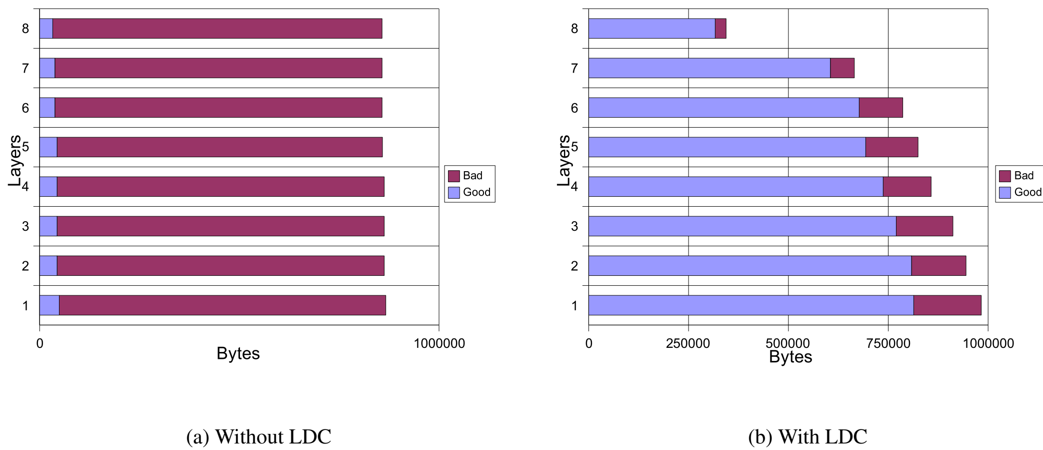
(a) 1024 Kbps (b) 512 Kbps (c) 256 Kbps

Figure 5. Received data after 60 seconds using TCP



(a) 1024 Kbps (b) 512 Kbps (c) 256 Kbps

Figure 6. Received data after 60 seconds using LDC



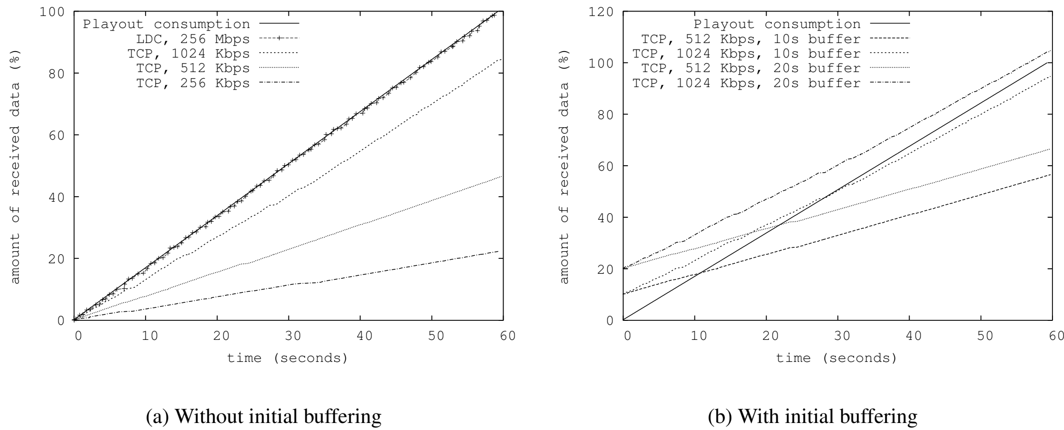
(a) Without LDC (b) With LDC

Figure 7. Achieved *goodput* with and without LDC

identical except that the amount of loss is decreased). In contrast, the application using the regular TCP has problems providing a continuous playout from the occurence of the first packet loss. Thus, LDC may also have some data arriving too late as we cannot control the retransmissions themselves, but in summary, the bandwidth is better utilized for useful data.

7 Comparison and Discussion

TCP is frequently used for time-dependent data streams because other transport protocols are often ignored and blocked. We are therefore looking at streaming over TCP although appropriate and useful mechanisms in or on top of other transport protocols exist. For example, using UDP, we



(a) Without initial buffering

(b) With initial buffering

Figure 8. Data arrival according to data consumption

would lose the same packets and not experience retransmission problems, giving us approximately the same gaps in the data playout that we experience using LDC. The reliability can also be achieved by adding user-space mechanisms on top of UDP or DCCP [7], or standards-track extensions to SCTP [9]. Nevertheless, the main problem is the required use of TCP, and we have therefore addressed how we can enhance the TCP API to better support time-dependent data streams.

We have compared the results from our LDC implementation with a standard Linux setup using New Reno with DSACK. We realize that several "high-speed" TCP variations available in Linux achieve higher throughput than New Reno with DSACK. However, the older New-Reno-based variants perform best regarding retransmission handling and latencies [4]. Furthermore, our LDC implementation does not influence fairness and congestion control (which is the main difference between the various other TCP versions) and can therefore be used with all the other TCP variants in Linux.

The test results show that an application can deliver a lot more relevant data with LDC than with regular TCP in a time-dependent scenario. In our example, LDC improved the perceived quality at the receiver by enabling a real-time playout of the video in the case of limited bandwidth and loss – with a quality according to the available rate. The data delivered by regular TCP is not very useful in this very strict real-time scenario. Data arrives too late according to the playout rate (figure 8(a)). If we were to implement a layered streaming architecture with regular TCP, we would have to check available resources before sending a layer, requiring larger buffers and introducing latencies, or using dynamically sized TCP send buffers [2].

Initial startup buffering, as used by many players today, would improve the situation. This is shown in figure 8(b) where the same scenario is plotted with a 10 seconds and a 20 seconds startup buffering. However, challenges here are the requirement for memory (especially at thin clients) and determining the size of the buffer, which should be dependent on the playout-, loss- and transmission rate (all of which can oscillate). Therefore, the test results clearly state that streaming of layer-encoded media greatly benefits from LDC support, both with and without buffering.

Another observation is that the throughput of TCP with LDC is slightly lower than without LDC. If we look only at the throughput, we get the values shown in table 1. We have not investigated why this is the case, but there can be several reasons. In the first place, data in the LDC packet ring is wrapped in `skbs` just in time before it is sent, and this may introduce an extra delay. In contrast, with regular TCP, we already have the data ready as `skbs` in the send buffer. Another reason may be our means of dropping packets. In our test applications, we have an LDC packet size equal to data for one layer in one second (128 kbit). So, when we are dropping packets, we may not fully utilize the available bandwidth. This may be tuned with different LDC packet buffer sizes. The important thing to remember is that even if we have slightly lower throughput with LDC than with the regular TCP implementation, all the data received with LDC support is usable, while that received with the regular implementation is not. A good throughput is useless if the data transmitted is too old.

Another property of the LDC implementation is that the memory area between the application and the kernel is shared. This enables a zero-copy data path where the `skbs` use the LDC send buffer's data pointer instead of just-in-time copying from user-space. The benefits of zero-copy approaches have been reported a numerous times before, recently in our study of enhancements to Linux 2.6 kernel [5]. We have not experimented with a CPU bound system and therefore not investigated this further for this paper.

It may be argued that the LDC API introduces complica-

tions because it lacks logic for dropping packets. LDC gives the means for dropping to the application but the application must itself implement a (good) drop algorithm to achieve useful results. However, the knowledge of the relevance of the data lies with the application (or programmer) and not the kernel. The adaptive send buffer size approach [2] may thus be simpler but the LDC API allows reacting even more quickly to congestion.

Our test includes only a layered video streaming scenario. However, the LDC functionality is useful for a larger class of applications. Applications that send data with a severely limited lifetime over TCP, like multiplayer games updates and multimedia sensor data may benefit from LDC support in TCP. Existing TCP-based applications can easily adopt LDC support while continuing to use their already implemented TCP architecture with minimal modification. After defining the application's socket as an LDC socket, there are only two modifications needed, that is the actual sending of packets and adding logic for dropping packets when they are outdated. As described below, one may want to implement functionality that lets the kernel drop packets automatically, making logic for dropping packets in the application unnecessary in the simplest cases.

To fully support delivery of time-dependent data over TCP, we would like to extend the drop functionality. Adding the timed reliability of PR-SCTP [10] would allow dropping by the kernel. Furthermore, as shown in figure 7(b), we still have some data arriving late due to retransmissions, i.e., the packets are processed and no longer controllable by LDC. Thus, it may also be desirable to include dropping of retransmissions, as these may be outdated due to the extra delay. However, in case of the PR-SCTP solution and the PRTP-ECN extension to TCP [3], this also implies a modification to the receiving side of a connection due to sequence numbers. A possible solution would then be to orthogonally use the solution of TCP Urel to send fresh data in every segment regardless of whether the segment is a new packet or a scheduled retransmission.

8 Conclusions

Many existing applications require real-time delivery of time-dependent data. TCP is used for this in spite of several problems. In this paper, we have therefore investigated an LDC extension to the sender-side handling of TCP in Linux. This allows applications to modify or drop queued but still unsent packets without any changes to the TCP protocol itself.

Our experiments with a layered video playout showed that we can improve the user experience in a scenario with time critical data in congested networks. Applications are able to send more useful data using LDC. Thus, the *perceived quality* for the receiver at a very low latency will be better since we manage to transfer the whole base layer and thereby get a continuous playback of the video in the described test scenario. We can for the same reason claim that we have a better utilization of the throughput, giving us a higher *useful throughput* (or *goodput*) with LDC than without. Based on this, we conclude that LDC support in TCP actually reduces the latency and increases the throughput for time critical data in congested networks.

We are currently performing more tests to further show the benefits of our LDC implementation. Further work on our system includes extending the drop functionality by adding both dropping after a deadline and some kind of selective retransmission.

References

[1] E. Birkedal. Late data choice with the linux TCP/IP stack. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, May 2006.

[2] A. Goel, C. Krasic, K. Li, and J. Walpole. Supporting low latency TCP-based media streams. In *Proceedings of the IEEE International Workshop on Quality of Service (IWQoS)*, pages 193–203, May 2002.

[3] K.-J. Grinnemo and A. Brunstrom. Enhancing TCP for applications with soft real-time constraints. In *Proceedings of SPIE Multimedia Systems and Applications*, pages 18–31, Nov. 2001.

[4] C. Griwodz and P. Halvorsen. The fun of using TCP for an MMORPG. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*. ACM Press, May 2006.

[5] P. Halvorsen, T. A. Dalseng, and C. Griwodz. Assessment of data path implementations for download and streaming. In *Proceedings of the International Conference on Distributed Multimedia Systems (DMS)*, pages 228–233, Sept. 2005.

[6] F. T. Johnsen, T. Hafsøe, C. Griwodz, and P. Halvorsen. Workload characterization for news-on-demand streaming services. In *Proceedings of the IEEE International Performance Computing and Communications Conference (IPCCC)*, Apr. 2007.

[7] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), Mar. 2006.

[8] J. Lai and E. Kohler. Efficiency and late data choice in a user-kernel interface for congestion-controlled datagrams. In *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, pages 136–142, Jan. 2005.

[9] L. Ong and J. Yoakum. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286 (Informational), May 2002.

[10] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758 (Proposed Standard), May 2004.