UNIVERSITY OF OSLO
Department of Informatics

Late data choice with the Linux TCP/IP stack

Master Thesis

Erlend Birkedal

24th May 2006

Abstract

The amount of time critical data sent on the Internet is increasing. As a result, the networks are in danger of getting congested. When sending time critical data on congested networks, we risk that the data may get too old in buffers, and are to no use when it arrives at its destination. For example, a late video frame in a video session or a late position update in games. To prevent delivery of outdated data, we want to discover and discard outdated network traffic as early as possible. A mechanism to accomplish this is *late data choice* (LDC). With LDC we address the problem at the source by taking control of our own sending buffer. This thesis explores the possibility to support LDC in TCP in the Linux network architecture. There is no LDC support in the TCP implementation in Linux today. Therefore, an implementation of LDC support for TCP was made and evaluated. When testing, we found that with LDC support in TCP we can deliver relevant data and drop data we no longer want to or can send. This makes the utilization of resources much better, as we do not waste them on useless data. Thus, we can say that LDC support in TCP reduces the *perceived latency* for the receiver, and increases *useful throughput*.

Contents

1 Introduction **1**
- 1.1 Background and motivation 1
- 1.2 Problem definition 1
- 1.3 Approach and method 2
- 1.4 Outline 2

2 The Linux Network architecture **3**
- 2.1 The Linux Operating System 3
  - 2.1.1 The Kernel 3
  - 2.1.2 Kernel Modules 3
- 2.2 The Network architecture 4
- 2.3 Receiving a TCP packet 4
  - 2.3.1 Datalink layer 4
  - 2.3.2 Network (IP) layer 6
  - 2.3.3 Transport layer 8
  - 2.3.4 Application layer 8
- 2.4 Sending a TCP packet 8
  - 2.4.1 Application layer 9
  - 2.4.2 Transport layer 10
  - 2.4.3 Network (IP) layer 12
  - 2.4.4 Datalink layer 12
- 2.5 In-depth study of TCP output 12
- 2.6 Summary 14

3 Late Data choice and related work **15**
- 3.1 Why and what? 15
- 3.2 Related work 16
  - 3.2.1 DCCP API for LDC 16
  - 3.2.2 SCTP Partial Reliability Extension 18
  - 3.2.3 Priority-Progress streaming 18
  - 3.2.4 Dynamically adjusting the TCP send buffer size 19
- 3.3 Summary 19

4 Late data choice support in TCP **21**
- 4.1 Design 21
  - 4.1.1 The packet ring 21

4.1.2 Replacing the existing socket buffer 22
4.1.3 Indexes and flags 23
4.1.4 Security 24
4.2 Implementation of TCP LDC in the Linux kernel 24
4.2.1 Modifications of the kernel 24
4.2.2 The TCP LDC kernel module 25
4.2.3 The TCP LDC user space library 30
4.2.4 Implementation choices and issues 31
4.3 Summary 33

5 Testing **35**
5.1 Test setup 35
5.1.1 TCP setup 36
5.1.2 TCP LDC setup 36
5.2 Results 37
5.2.1 TCP results 38
5.2.2 TCP LDC results 38
5.2.3 Comparison 40
5.3 Discussion 40
5.4 Summary 41

6 Conclusion and further work **43**
6.1 Summing up 43
6.2 Conclusion 43
6.3 Further work 44

Appendices **45**

A Source code **45**
A.1 tcp_set_ldc() 45
A.2 ldc_pkt_buffer_alloc() 46
A.3 ldc_notify() 49
A.4 ldc_next_pkt() 49
A.5 ldc_update_send_head() 50

B The CD-ROM **53**

List of Figures

2.1 Structure of socket buffers (`struct sk_buff`) [14]. 4
2.2 Packet reception [11]. 5
2.3 Network layer data path [11]. 7
2.4 TCP input processing. [11] 9
2.5 TCP output processing [11]. 11
2.6 Transmission of a packet [11]. 13
2.7 Function graph over a part of TCP output. 14

3.1 Empty packet ring [9]. 16
3.2 User adds packets, shifting user_i and umod_i. Packets 0-6 are owned by the kernel [9]. 16
3.3 Kernel processes packets 0 and 1 and sends them to the device, shifting kern_i [9]. 17
3.4 User marks packet 1, 3 and 5 as dead. Note that no index movement is needed and marking packet 1 as dead is safe [9]. 17
3.5 User wants to alter previously-sent packets, moves umod_i. Packets 3-6 are safe to modify [9]. 17
3.6 Like figure 3.5, but the kernel moved kern_i simultaneously with the user moving umod_i. Only packets 5 and 6 are safe to modify [9]. 17

4.1 TCP LDC design 22
4.2 TCP LDC packet ring and indexes 23
4.3 Mapping memory into the application's memory space 27
4.4 User space memory in kernel context 32

5.1 Test setup 36
5.2 Layered video 36
5.3 TCP test results 37
5.4 TCP LDC test results 39
5.5 Results from an application that used LDC improperly 41

List of Tables

2.1 Receive functions and their source files 6
2.2 Send functions and their source files 10

4.1 Functions mentioned in section 4.2 and their source files 25

5.1 Receive rate 39

Chapter 1

Introduction

1.1 Background and motivation

We send more and more data in our computer networks. The amount of time critical data is also increasing, i.e., especially due to the fact that distributed interactive applications like online games and high rate streaming of audio and video gain ground as more people gets a broadband connection to the Internet at home. As a result, the networks are in danger of getting congested, and peers may receive data that has gotten too old in the buffer and therefore are of no value to the receiver. This is because when networks get congested, data may be delayed in buffers along the path. Packets may even get dropped and retransmitted if a reliable connection, like Transmission Control Protocol (TCP), is used. This is just fine for time-independent data. For time-critical data on the other hand, the situation is different. If the network is congested and time critical data is delayed, we risk that the data may get too old, and are to no use, when it arrives at its destination. As an example, consider a late video frame in a video session or a late position update in games. As a result, the whole transmission of the data was useless. To prevent delivery of outdated data, and even free bandwidth, it would be nice if one could discover and discard network traffic already too old as early as possible. A mechanism to accomplish this is *late data choice* (LDC) [9]. With LDC we address the problem at the source by taking control of our own sending buffer. There is no LDC support in the TCP implementation in Linux today.

1.2 Problem definition

We want to utilize the network resources on the most relevant data, e.g., in games or video streams, the relevancy of data rapidly decrease over time. With LDC we can drop and discard packets that has not been sent before the payload is too old. For MPEG video streams, one could choose to send the important I-frames and drop less important B and P-frames if the network is congested and the data got old in the buffer. In this thesis, we want to explore the possibility to reduce latency and congestion with LDC support for TCP for time sensitive applications, such as interactive online games or video conferencing. We want to see if we can reduce latency and congestion by not sending old data unnecessarily. By doing this we want to see if we can send more usable data because resources that ordinarily would be used on outdated data, can instead be used on new and useful data. LDC is a modification on the sending side of a connection and should be transparent to the receiver, i.e. it should not be necessary to alter the receiving side to make LDC work. Thus, we address the question of;

> *Can late data choice support in TCP reduce latency and increase throughput for time critical data in congested networks?*

1.3 Approach and method

The work on this thesis started with an investigation of the Linux network architecture. One need to have a knowledge of mechanisms and structures used in the network architecture to be able to extend it with new functionality. To get greater insight into LDC, related work was studied. A design of LDC support in TCP was done and implemented. Finally, the implementation was evaluated and compared with TCP without LDC support. The results show that TCP with LDC support can deliver much more relevant data within given time limits. Thus, giving a lower *perceived latency* and higher *useful throughput*.

1.4 Outline

In chapter 2 we describe how the Linux network architecture, with main focus on TCP/IP, is implemented. We follow a packet up and down the TCP/IP stack and take a look at what happens with the packet along the path. Next, in chapter 3, the LDC mechanism is described in greater detail. We also take a look at some related work in the field of LDC and partial reliability. Then, in chapter 4, the LDC support in TCP is presented. First the design is discussed, and in the following section, the implementation is described in detail. In chapter 5, an evaluation of TCP with and without LDC is described, and the test results are compared and discussed. Finally, the conclusion is presented in chapter 6, as well as ideas for further work and development of LDC with TCP is discussed. There are also two appendices. Appendix A lists an extract of the source code of the LDC for TCP implementation. Appendix B describes the attached CD-ROM. The CD-ROM contains all the source code for this thesis.

Chapter 2

The Linux Network architecture

This chapter starts with a brief introduction to the Linux Operating System. Then the TCP/IP implementation in the Linux Kernel is explained by following the packet trail from when a packet enters the network interface card (NIC), passes up and down the network stack and leaves the NIC again. Details on how TCP connections are set up and torn down will not be explained, but we will mention it when appropriate. The kernel version used is 2.6.15.4 [10]. As guide trough the Linux network architecture, we used sources [14] and [11]. To browse the kernel code, the great code browsing tool LXR [3] was used.

2.1 The Linux Operating System

Linux is an open source UNIX-like operating system (OS) originally written by Linus Torvalds from Finland. Now, it is a fully functional OS used all around the world, used for desktop and server systems by both individuals and businesses. Open source means that the actual source code is open for the public and free to use and modify. Because Linux is open source and easily modified, it has become very popular among students and lecturers in the computer science departments of universities around the world. With a widely spread and stable OS like Linux, you can easily implement and test new technologies.

2.1.1 The Kernel

The Linux kernel is a monolithic OS kernel. This means that the whole kernel is loaded into memory at system start and all kernel code run in kernel mode (privileged mode). In contrast, a micro-kernel (like Windows NT) is very small and do only the basics like memory management. Other functionality runs in user mode (restricted mode). The fact that Linux is monolithic, made it very big and hard to maintain. To solve this problem, the Linux developers started to use kernel modules.

2.1.2 Kernel Modules

Kernel modules are code that do a specific job. It may be a kernel modification, enhancement or a device driver. Unlike in a micro-kernel system, this additional code

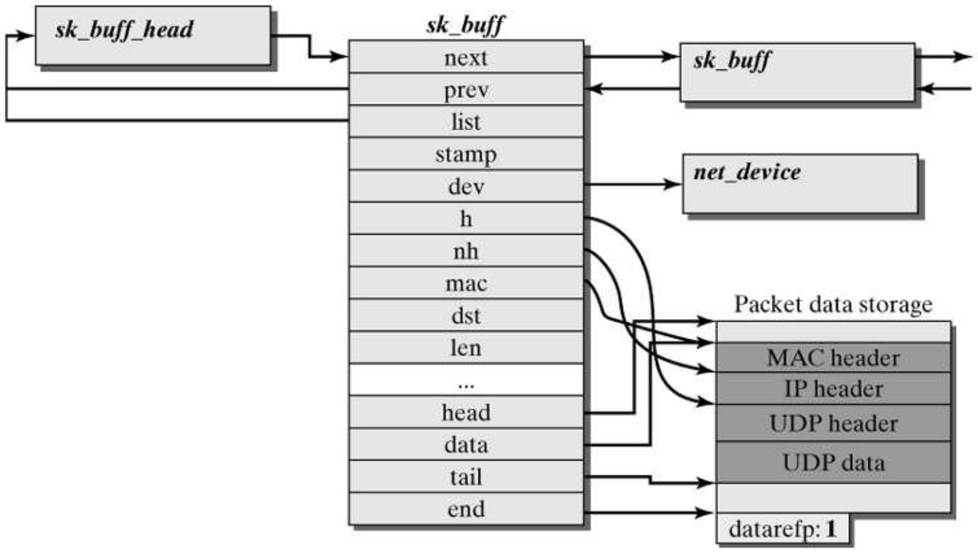


Figure 2.1: Structure of socket buffers (`struct sk_buff`) [14].

does not run in user mode, but it is loaded into the kernel memory and runs in kernel mode. This leads to fewer context switches than in an micro-kernel system, since the code in modules do not need to issue system-calls to interact with the kernel. The additional code in micro-kernel systems have to issue system-calls, as they run in user mode.

2.2 The Network architecture

In the Linux kernel, all packets are represented by the socket buffer `struct sk_buff` (`skb`), see figure 2.1. As we can see in the figure, `skb`s are connected in a linked list with a list head, `sk_buff_head`, on the top. This makes it a trivial task to traverse all `skb`s in a buffer. The `skb` is passed from layer to layer along the network stack as it gets processed on the way in and/or out. In the next sections we will see how the Linux kernel handle the network packets as `skb`s, starting when a packet arrives at the NIC.

2.3 Receiving a TCP packet

In this section we describe the journey of an incoming packet, starting at the NIC. When the NIC has picked up a packet from the wire and is ready to pass it to the OS, it issues a hardware interrupt, and the data link layer starts processing the packet.

2.3.1 Datalink layer

The interrupt issued at the arrival of a packet, is checked by the interrupt-handling routine of the NIC's network driver, so that the appropriate action can be taken. If the

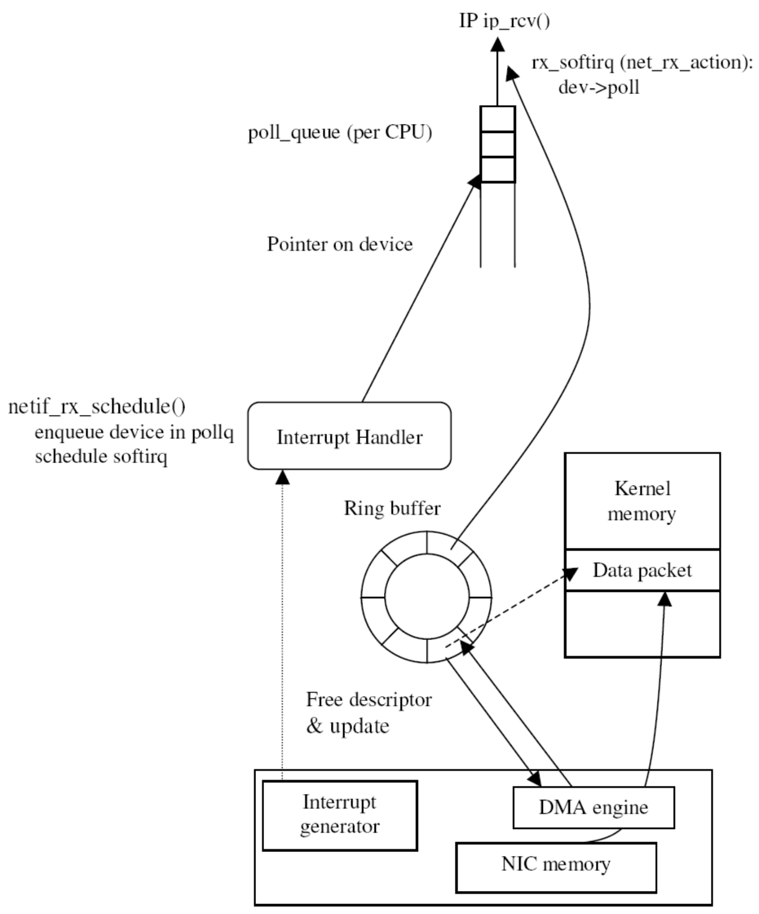


Figure 2.2: Packet reception [11].

interrupt issued because of an incoming packet, it calls on `net_rx()` (driver dependent) for further handling. See table 2.1 for functions and source files mentioned in this section. The `net_rx()` function uses `dev_alloc_skb()` to allocate an `skb`. Then, as we can see in figure 2.2[1], the packet data is copied from NIC memory to the socket buffer in kernel memory. The pointer `skb->dev` is set to the receiving NIC. Then, `netif_rx_schedule()` is called to complete the interrupt handling. Here, a reference to the device (the receiving NIC) is queued on the (current) CPU's `poll_list`, `softnet_data->poll_list`. Finally a soft interrupt, `NET_RX_SOFTIRQ`, is raised.

After a certain time interval, the kernel checks if there is any soft interrupts that needs handling, and if so, the `do_softirq()` is called. The CPU then polls the devices present in its `poll_list` and calls on `poll()` (driver dependent) for each device. The drivers `poll()` function then calls `netif_receive_skb()`. If the packet is an IP-packet `netif_receive_skb()` will call `ip_rcv()` and the network layer will continue the processing.

Function	**Source file**
`dev_alloc_skb()`	include/linux/skbuff.h, line 1070
`icmp_rcv()`	net/ipv4/icmp.c, line 926
`igmp_rcv()`	net/ipv4/igmp.c, line 863
`ip_forward()`	net/ipv4/ip_forward.c, line 57
`ip_local_deliver()`	net/ipv4/ip_input.c, line 266
`ip_local_deliver_finish()`	net/ipv4/ip_input.c, line 200
`ip_mr_input()`	net/ipv4/ipmr.c, line 1335
`ip_rcv()`	net/ipv4/ip_input.c, line 376
`ip_rcv_finish()`	net/ipv4/ip_input.c, line 334
`ip_route_input()`	net/ipv4/route.c, line 2083
`netif_receive_skb()`	net/core/dev.c, line 1579
`netif_rx_schedule()`	include/linux/netdevice.h, line 833
`tcp_rcv_established()`	net/ipv4/tcp_input.c, line 3725
`tcp_rcv_state_process()`	net/ipv4/tcp_input.c, line 4219
`tcp_v4_do_rcv()`	net/ipv4/tcp_ipv4.c, line 1138
`tcp_v4_rcv()`	net/ipv4/tcp_ipv4.c, line 1189
`tcp_v4_send_reset()`	net/ipv4/tcp_ipv4.c, line 676
`udp_rcv()`	net/ipv4/udp.c, line 1118
`do_softirq()`	kernel/softirq.c, line 116

All line numbers refers to an unchanged 2.6.15.4 source tree.

Table 2.1: Receive functions and their source files

2.3.2 Network (IP) layer

When `ip_rcv()` receives a packet from the underlying layer, it checks if the size of the packet is at least as big as an IP header, if it is an IPv4 packet, if the checksum is correct and if the packet has the correct length. If any of the tests fail, then the packet is dropped. At this point, the netfilter hook `NF_IP_PRE_ROUTING` is invoked, as we can see on the left in figure 2.3. After any procedures at the hook have finished, `ip_rcv_finish()` is called.

In `ip_rcv_finish()`, it is determined where the packet should travel further. This is done by `ip_route_input()` setting the `skb->dst`. `ip_rcv_finish()` also checks the packet for any IP options and creates the `ip_options` struct if needed. Now it is time to send along the packet by calling `skb->dst->input()`. It has been set to one of the following by `ip_route_input()` (see figure 2.3):

- `ip_local_deliver()` - Unicast and multicast packets for local delivery.
- `ip_forward()` - Unicast packets that should be forwarded.
- `ip_mr_input()` - Multicast packets that should be forwarded (not shown in the figure).

[1]Figures 2.2 through 2.6 are taken from [11]. The kernel version in the figures and the one presented here are not the same, so some minor differences may occur.

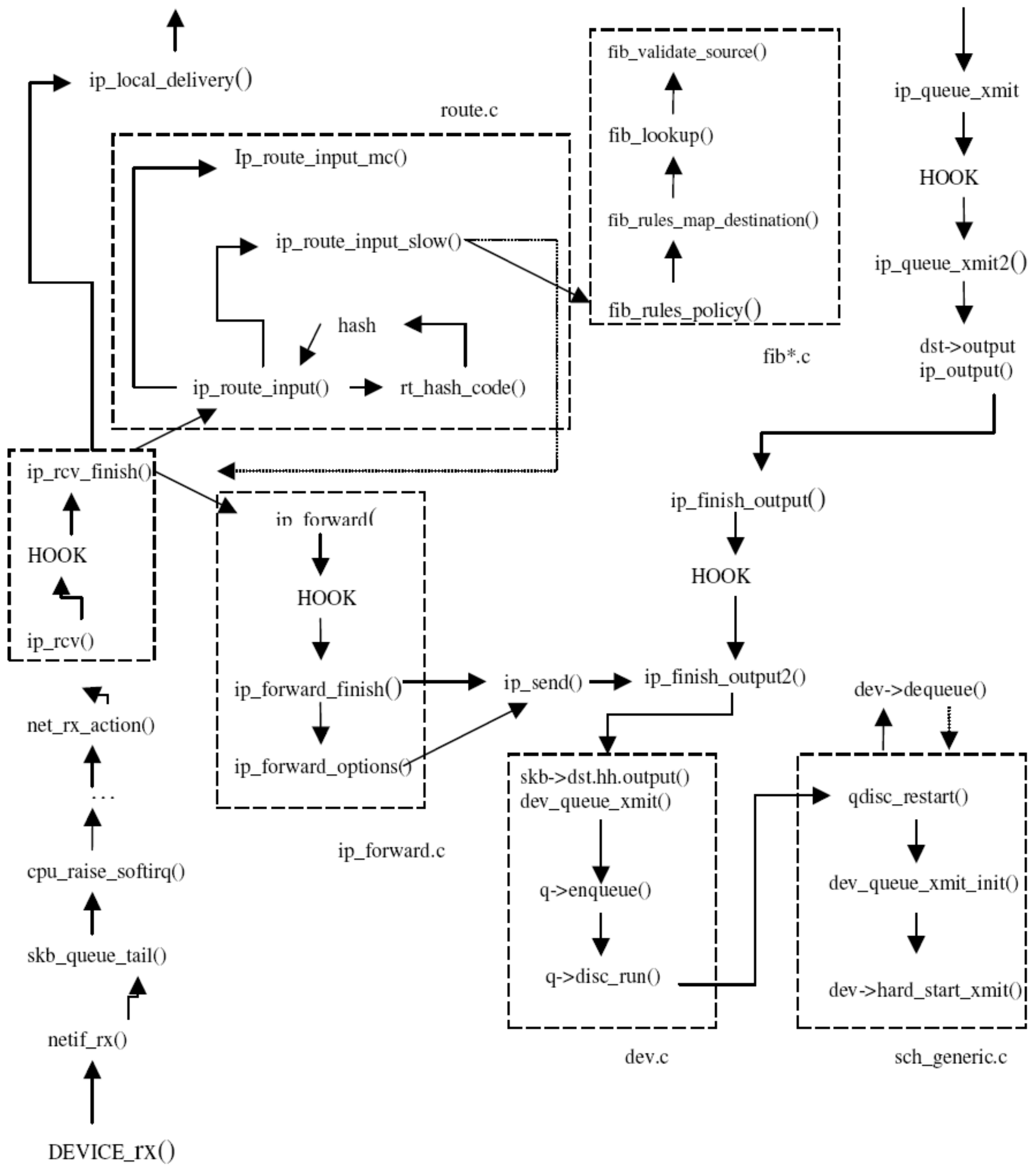


Figure 2.3: Network layer data path [11].

In this description, we assume that the packet was for local delivery, i.e., `ip_local_deliver()` was called. The only thing `ip_local_deliver()` does is to reassemble IP fragments. After that is done, the netfilter hook `NF_IP_LOCAL_IN` is invoked. When it returns, `ip_local_deliver_finish()` is called.

At `ip_local_deliver_finish()`, it is determined if the packet is intended for a RAW-IP socket. If not, the transport protocol for further processing has to be determined. This is done by choosing a transport protocol from the hash table `ipprot`, using the protocol ID of the IP header modulo (`MAX_INET_PROTOS -1`) as the hash value. These are the most common handling routines for the transport layer:

- `tcp_v4_rcv()` - Transmission Control Protocol

- `udp_rcv()` - User Datagram Protocol
- `icmp_rcv()` - Internet Control Message Protocol
- `igmp_rcv()` - Internet Group Management Protocol

If the packet is not sent to a RAW-IP socket or a transport protocol could not be found, the packet is dropped and an ICMP Destination Unreachable message is returned to the sender. In this example, we assume further that the packet is a TCP packet and the `tcp_v4_rcv()` function will be called to process the packet at the transport layer.

2.3.3 Transport layer

First, `tcp_v4_rcv()` checks if the packet it got really is for this host. Otherwise, it is discarded. If the packet was not discarded, it is checked if this packet belongs to an active socket. If an active socket was found, `tcp_v4_do_rcv()` is called. If not, `tcp_v4_send_reset()` is called to send a RESET segment.

On the left in figure 2.4, we can see `tcp_v4_do_rcv()` and paths it can take from there. `tcp_v4_do_rcv()` checks the state of the connection. We assume that it is established, `TCP_ESTABLISHED`, and that `tcp_rcv_established()` is called. If the connection was not established, `tcp_rcv_state_process()` is called to handle the connection (setup, tear down etc.).

`tcp_rcv_established()` distinguishes between two paths:

- *Fast path* is used for trivial packets, that are either pure ACKs, or if it contains data expected next (Header Prediction). It is used for most of the packets (83% - 100%) [14] and speeds up the performance.
- *Slow path* handles all the packets not sent the fast path. It has more functionality to handle all the exceptions.

If the packet took the fast path and is a data packet, the payload is now copied to the receive memory of the process that owns the connection. Packets in sequence go to the `sk_receive_queue` and out of sequence packets to the `out_of_order_queue`. At last, the process is notified (sk->sk_data_ready).

2.3.4 Application layer

The applications can now read (e.g. `read()`) data from their sockets and use it as they want. There may still be more protocol handling to do if a application layer protocol like HTTP [1] or RTP [5] is used.

2.4 Sending a TCP packet

We will now look at the operations for sending a packet using TCP and describe its path through the system.

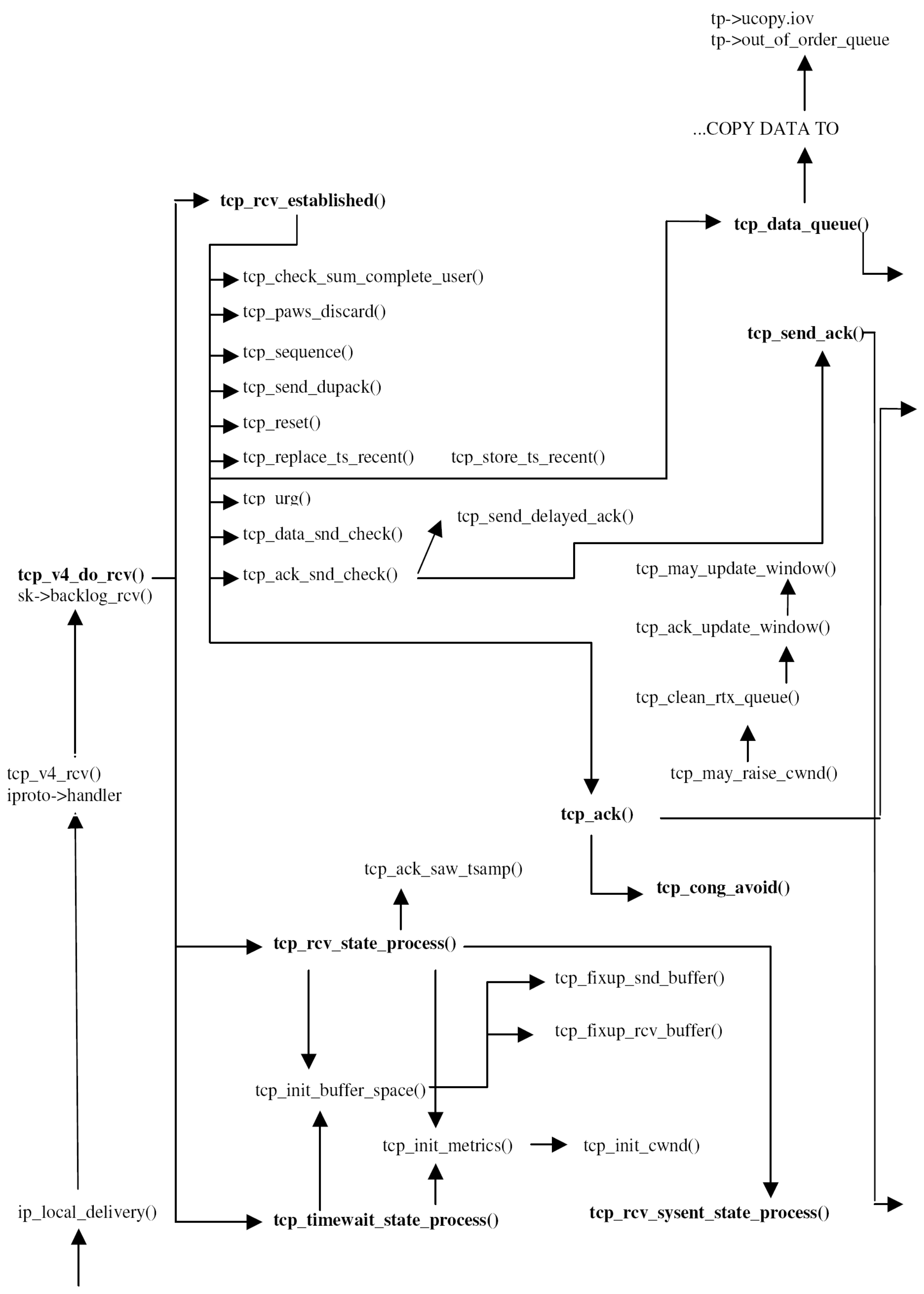


Figure 2.4: TCP input processing. [11]

2.4.1 Application layer

Applications use a system-call to send payload over sockets, e.g., `send()`. The system-call invokes the `tcp_sendmsg()` function to send TCP packets. See table 2.2 to see

Function	Source file
`dev_queue_xmit()`	net/core/dev.c, line 1245
`ip_finish_output()`	net/ipv4/ip_output.c, line 203
`ip_finish_output2()`	net/ipv4/ip_output.c, line 163
`ip_fragment()`	net/ipv4/ip_output.c, line 414
`ip_output()`	net/ipv4/ip_output.c, line 274
`ip_queue_xmit()`	net/ipv4/ip_output.c, line 285
`qdisc_restart()`	net/sched/sch_generic.c, line 93
`qdisc_run()`	include/net/pkt_sched.h, line 222
`sk_stream_wait_connect()`	net/core/stream.c, line 51
`skb_realloc_headroom()`	net/core/skbuff.c, line 675
`tcp_current_mss()`	net/ipv4/tcp_output.c, line 680
`tcp_fragment()`	net/ipv4/tcp_output.c, line 448
`tcp_push()`	net/ipv4/tcp.c, line 488
`tcp_push_pending_frames()`	include/net/tcp.h, line 878
`tcp_sendmsg()`	net/ipv4/tcp.c, line 662
`tcp_snd_wnd_test()`	net/ipv4/tcp_output.c, line 840
`tcp_transmit_skb()`	net/ipv4/tcp_output.c, line 265
`tcp_write_xmit()`	net/ipv4/tcp_output.c, line 999

All line numbers refers to an unchanged 2.6.15.4 source tree.

Table 2.2: Send functions and their source files

functions mentioned in this section and their source files.

2.4.2 Transport layer

`tcp_sendmsg()` copies the payload from user-space memory to kernel memory and sends the payload as TCP segments. As for incoming packets, the way this is handled varies on the state of the connection. We assume that it is established, `TCPF_ESTABLISHED`. If the connection was not established, the `sk_stream_wait_connect()` would be called and further handling would wait until we got a connection. The next thing `tcp_sendmsg()` does, is to calculate the maximum segment size (MSS) by calling `tcp_current_mss()`. Then, the actual copying of data is done. Data from user-space is put into socket buffers and added to the end of the socket's transmit queue. The packet is then sent by calling `tcp_push()`, which in turn calls `tcp_push_pending_frames()`. See the call sequence in figure 2.5. `tcp_write_xmit()`, called from `tcp_push_pending_frames()`, checks if the packet needs to be fragmented. If it does, `tcp_fragment()` is called. `tcp_write_xmit()` calls `tcp_transmit_skb()` as long as it is allowed to do so by `tcp_snd_wnd_test()`.

`tcp_transmit_skb()` is completing the TCP segment by building the TCP header and checksum it. When done, `tp->af_specific->queue_xmit()`, which is a pointer to the underlying protocol's service access point (SAP), is

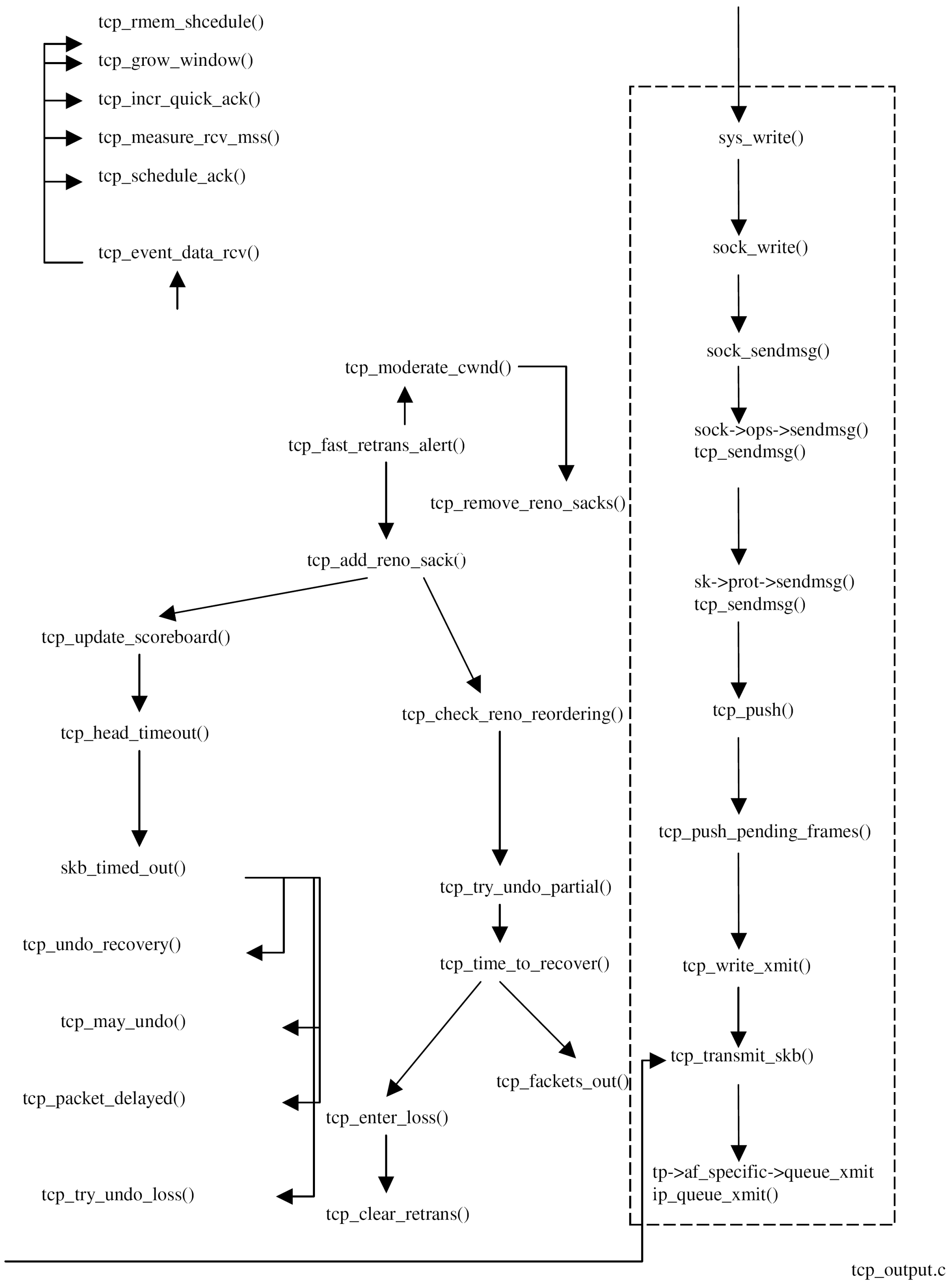


Figure 2.5: TCP output processing [11].

called to pass the socket buffer to the IP layer for transmission. In this case `tp->af_specific->queue_xmit()` refers to `ip_queue_xmit()`.

2.4.3 Network (IP) layer

As we can see in figure 2.3, the path through the network layer starts at ip_queue_xmit() when it receives a socket buffer from the transport layer. It checks if skb->dst includes a pointer to an entry in the routing cache. If it does not find a suitable route, the necessary steps to setup a route is taken. We assume that a route exists. When it is determined where that packet should be sent, the IP header is allocated and built. At the end of ip_queue_xmit(), the netfilter hook NF_IP_LOCAL_OUT is invoked. After it has finished, it continues at skb->dst->output(), that is a function pointer that is set during the routing process and points to ip_output() for IP packets. ip_queue_xmit2() shown in the figure does no longer exist.

Here, the packet size is checked. If the size is greater than the MSS, ip_fragment() is called, otherwise ip_finish_output(). In this example, we assume no fragmentation is needed.

At this point, the layer-2 packet type is set to ETH_P_IP and a new netfilter hook, NF_IP_POST_ROUTING, is invoked and continues at ip_finish_output2().

At ip_finish_output2(), the IP handling of packets is finished. First, it checks if there is enough free space for the hardware header. If not, it allocates head room with skb_realloc_headroom(). Then, it checks if the skb already has a reference to the *hard header cache*. If it has, it copies the layer-2 packet header into the packet-data space, in front of the IP header. At the end it calls hh->hh_output(skb), which in this case is a pointer to dev_queue_xmit(). If there was no entry in the *hard header cache*, the corresponding address-resolution routine is invoked.

2.4.4 Datalink layer

For each packet dev_queue_xmit() gets, it grabs the device's queue (qdisc) and enqueues the socket buffer on it, seen in figure 2.6. Then, it calls qdisc_run(). qdisc_run() calls qdisc_restart() as long as the NIC's queue is not stopped (e.g. due to link failure or the tx_ring being full) and qdisc_restart() returns a value less than 0. In qdisc_restart(), packets get dequeued, and the virtual function hard_start_xmit() (driver dependent) is called. hard_start_xmit(), implemented in driver code, places the packet description in the tx_ring and the drivers tells the NIC that there are packets to send.

When the NIC has transmitted the packets, it (the driver) tells the CPU, with a soft interrupt, that packets are sent. The packets are put into a completion_queue and the memory used by the packet data is scheduled for deallocation.

2.5 In-depth study of TCP output

In this section, we will explain the areas where the LDC implementation will alter the current implementation. This area is mainly a part of the TCP output code in the kernel. As we are going to take a closer look at, in chapter 4, we want to replace the existing send buffer.

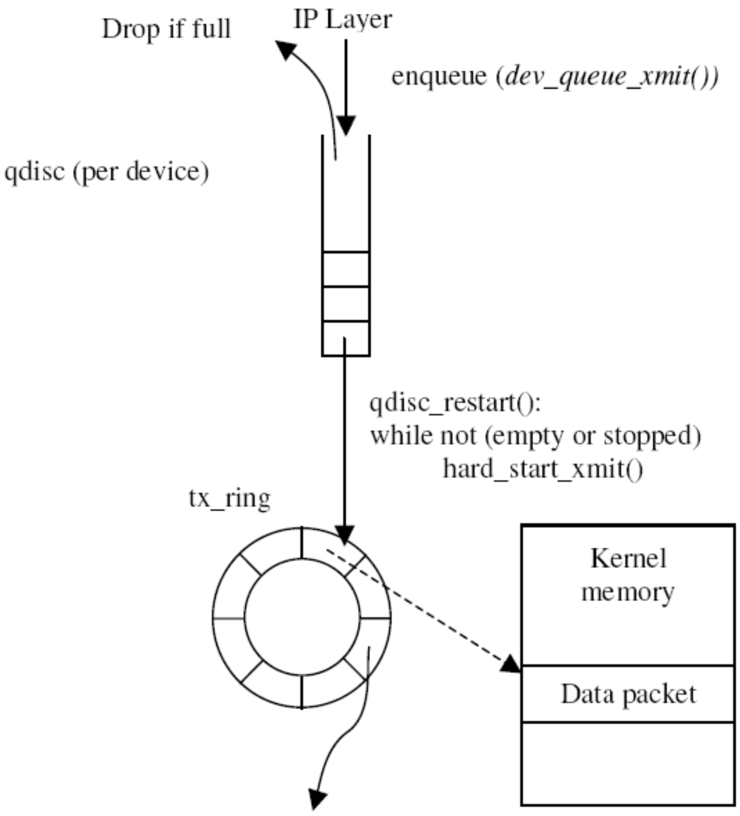


Figure 2.6: Transmission of a packet [11].

All data enters the send buffer via `tcp_sendmsg()` (or `do_tcp_sendpages()`). Here, data is copied from user space and put in an `skb`, and the `skb` is added at the tail of the send buffer. See figure 2.7 for the call sequence of the functions mentioned in this section. Both `tcp_write_xmit()` and `tcp_push_one()` take the `skb` at the socket's `sk_send_head`, if any, and pass it along to `tcp_transmit_skb()` for sending. The difference between these two, is that `tcp_push_one()` (as the name imply) only send one frame while `tcp_write_xmit()` send as long it has data in the buffer or is allowed to send by the congestion control mechanisms. As for function calls, you have one call to `tcp_transmit_skb()` and then one call to `update_send_head()` in `tcp_push_one()`, and a loop with calls to the two same functions in `tcp_write_xmit()`. `update_send_head()` sets the socket's `sk_send_head` to the next `skb` in the send buffer or `NULL` if empty. To summarize a bit, we have `tcp_sendmsg()` and equivalents as producers, and `tcp_write_xmit()`,`tcp_push_one()` and equivalents as consumers.

If sending of packets has stopped due to not enough space in the TCP send window, it can not continue to send before the window space is opened. As we know, the window opens when we get ACKs from the other side. ACKs (and other incoming packets) are, in an established state, handled by `tcp_rcv_established()`. As we see in figure 2.7, `tcp_rcv_established()` calls on `tcp_ack()` followed by a call to `tcp_data_snd_check()` when it receives an ACK. `tcp_data_snd_check()` tries to send data in the send buffer, ending up in the transmit-update head loop in

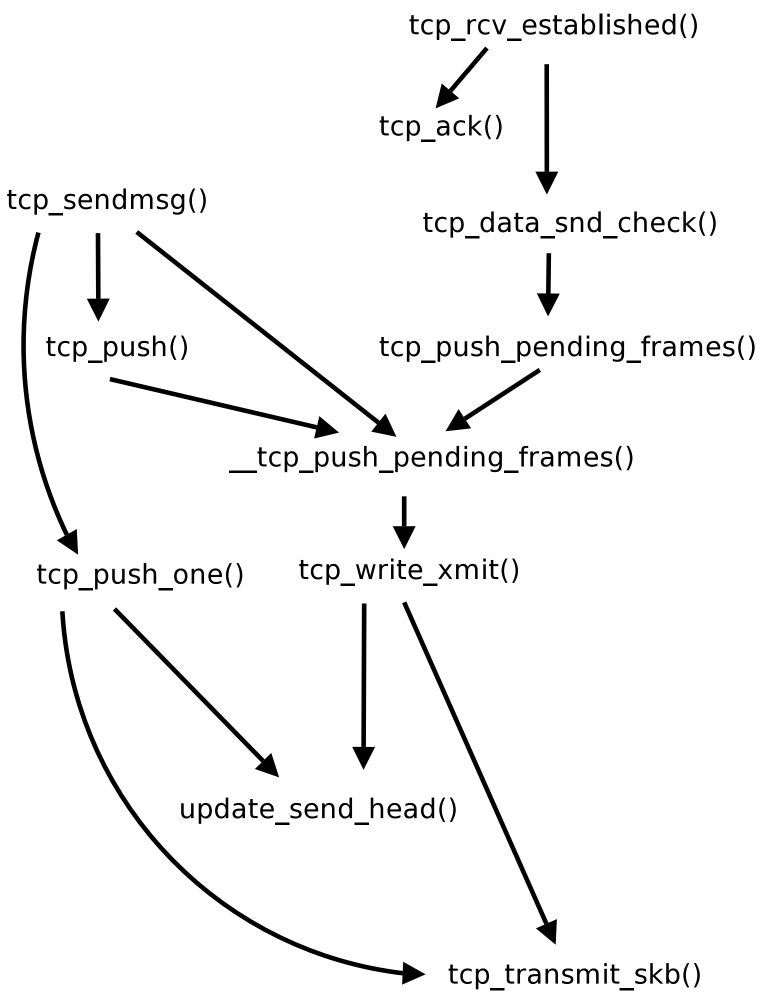


Figure 2.7: Function graph over a part of TCP output.

`tcp_write_xmit()`.

As we can see, there are many functions interacting with the send buffer. Details on how LDC fits into all this is described in detail in chapter 4, but in short, the send buffer is not entirely replaced. Packets are moved from the LDC buffer to the existing send buffer right before they are to be sent, i.e., in `tcp_sendmsg()` and `update_send_head()`.

2.6 Summary

In this chapter, we have given a brief introduction to the Linux OS. We have looked at the TCP/IP implementation in Linux by following a packet up and back down the TCP/IP stack. The parts of the stack that the LDC implementation will touch are explained in greater detail. By going into the Linux network architecture, we found that TCP currently has no mechanism to drop expired data packets. In the next chapter, we look at the possibility to discover and drop packets that has content that is invalid due to time restrictions.

Chapter 3

Late Data choice and related work

In this chapter, the concept of *late data choice* (LDC) is explained. Also, earlier work done in the field of late data choice and partial reliability is presented.

3.1 Why and what?

The reason to have control over the transmission buffer is to have the ability to ensure that we send the most relevant data at the moment. In general, this mean that we do not want to use resources on data that for some reason (e.g. congested network) has not been transmitted yet, and will have no value to the receiver if transmitted. For instance, in games or video streams, the relevancy of data rapidly decrease over time. With mechanisms like LDC you may remove, or update the content of, packets that have not been sent before the payload is too old. For video streams, one could choose to send the important I-frames and drop less important B and P-frames if the network is congested, and for games outdated events can be dropped.

LDC is the property to provide a generic control over the transmit buffer [9]. When an application has control over its transmit buffer, it has the ability to modify or remove data in the buffer.

A related term is *partial reliability* which is defined as a partially reliable 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" [12]. For instance, partial reliability in SCTP is a mechanism that allows a sender to signal to its peer that it should no longer expect to receive one or more data chunks. The other part to partial reliability is to give packets a parameter that states how long the data it contains is valid. This is called *timed reliability*.

The main difference between *LDC* and *partial reliability*, is when you need to decide the validity of packet data. With partial reliability you set a time to live on your data when you send a packet (i.e., put data in the transmit buffer). With LDC you do not set any time to live on the packet while sending, but you can remove it from the transmit buffer later. LDC may be more flexible than partial reliability, since you have direct access to the transmit buffer. With LDC, you do not just have the ability to drop a packets, but you can alter the data directly in the buffer if you want to, or maybe just remove a part of the packet data. Thus, partial reliability does not give the same

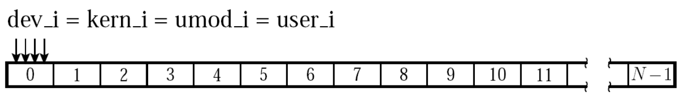


Figure 3.1: Empty packet ring [9].

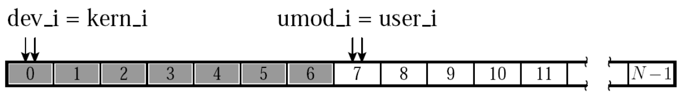


Figure 3.2: User adds packets, shifting user_i and umod_i. Packets 0-6 are owned by the kernel [9].

flexibility and dynamic behavior as the LDC approach, but gives the ability to not send outdated packets.

3.2 Related work

In this section, we take a look at some related work that has mechanisms to handle time critical data. The DCCP API for LDC [9] lets applications remove packets directly from the send buffer. The SCTP Partial Reliability Extension [13] lets applications give its packet a time to live. Another mechanism, Priority-Progress streaming [8], does prioritized sending of video in adaption windows, allowing to scale the send rate after available bandwidth. Finally, a solution to prevent data from getting old in the send buffer by dynamically adjusting the TCP send buffer size is presented [4].

3.2.1 DCCP API for LDC

The Datagram Congestion Control Protocol (DCCP) [7] is a unreliable transport layer protocol. It provides bidirectional unicast connections of congestion-controlled unreliable datagrams. It is suitable for applications that transfer fairly large amounts of data and that can benefit from control over the tradeoff between timeliness and reliability. To add support for LDC, the DCCP API for LDC [9] was developed.

The foundation in this solution is the packet ring. It is a shared memory segment that applications use to send packets (put packet records on), and the kernel uses to fetch packets to send to the NIC. The packet ring is an array of packet records and has four indexes; two for enqueued and empty, and two for late packet modification, see figure 3.1.

The packet records are data structures representing packets. They consists of a pointer to the actual packet data, the size and some flags. The *user_dead* flag is used by the applications to mark the packet as dead. If the flag is set, the kernel will elegantly skip the packet and move to the next. When using this technique to delete packets, the application does not need to remove the packet and shift other packets in the ring. The flag *kern_acked* is set when the kernel has got an ACK on a sent packet. The flag is used to inform the application that the packet is successfully sent. The last flag is named *user_seq* and is a local sequence number. If the application sets this flag, the kernel can inform the application when or if a packet has been ACKed even if the packet ring has wrapped and the *kern_acked* flag is overwritten.

As already mentioned, the packet ring has four indexes. Two owned by the kernel and two owned by the application. The two kernel indexes are *dev_i* and *kern_i*.

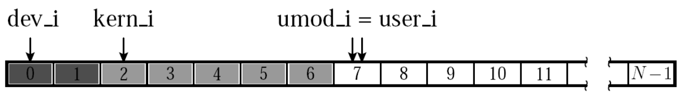


Figure 3.3: Kernel processes packets 0 and 1 and sends them to the device, shifting kern_i [9].

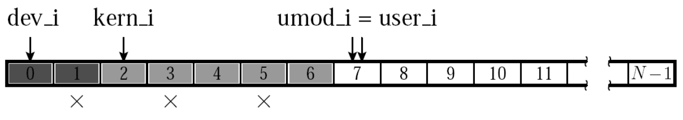


Figure 3.4: User marks packet 1, 3 and 5 as dead. Note that no index movement is needed and marking packet 1 as dead is safe [9].

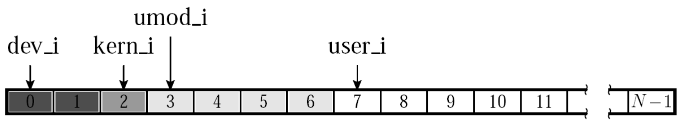


Figure 3.5: User wants to alter previously-sent packets, moves umod_i. Packets 3-6 are safe to modify [9].

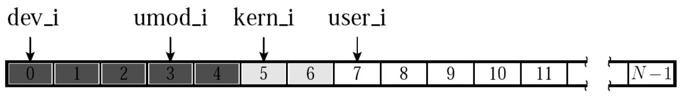


Figure 3.6: Like figure 3.5, but the kernel moved kern_i simultaneously with the user moving umod_i. Only packets 5 and 6 are safe to modify [9].

The device index, *dev_i*, points at the oldest packet record enqueued for device transmission. The kernel index, *kern_i*, points at the oldest packet record that the kernel has not processed yet. *umod_i* and *user_i* are the two application indexes. The user modification index, *umod_i*, is the only index that can move backwards. It points at oldest packet record that the application can safely modify. The user index, *user_i*, points immediately above the newest packet record that application has put on the ring. A packet with index *i*, where $dev\_i \leq i < kern\_i$, has been processed by the kernel and is waiting to be sent by the NIC. The application should not modify the packet. When $kern\_i \leq i < umod\_i$, the packet is available for kernel processing, but it has not been sent to a NIC. The application should not modify the packet without moving the *umod_i* backwards. Figures 3.1 through 3.6 show several examples of packet rings.

The API was implemented by Lai and Kohler as a part of their experimental in-kernel DCCP protocol implementation that runs in Linux 2.4.20. In their prototype, the buffers that contained actual packet data was allocated from shared memory between user space and kernel space that they called *packet zones*. For sending data "direct from disk" (e.g., a video stream), they loaded data directly into a packet buffer using raw disk or raw file support to avoid unnecessary copying of data.

Their hypothesis was that the packet ring used with DCCP would give low latency and high throughput in addition to LDC. Tests showed that applications that used the copy-free API outperformed comparable UDP applications by up to 60%. By reducing user-kernel crossings, it can further improve the performance by up to 71%. If you run a MPEG streaming server, you will get congestion control, low packet delay and LDC at the same time. The LDC enables the MPEG server to deliver more important frames (I-frames), as it can choose to drop less important frames (B and P-frames).

3.2.2 SCTP Partial Reliability Extension

Stream Control Transmission Protocol (SCTP) [13] is a reliable transport layer protocol protocol. It offers the following services to its users: Acknowledged error-free non-duplicated transfer of user data, data fragmentation to conform to discovered path MTU size, sequenced delivery of user messages within multiple streams, with an option for order-of-arrival delivery of individual user messages, optional bundling of multiple user messages into a single SCTP packet, and network-level fault tolerance through supporting of multi-homing at either or both ends of an association.

The SCTP Partial Reliability Extension (PR-SCTP) [12] is, as the name states, a partial reliability extension to the SCTP protocol. It provides mechanisms that allows an SCTP endpoint to signal to its peer that it should move the cumulative ACK point forward. If you have an SCTP connection where both sides support this extension, it can provide a partially reliable data transmission service. One example of a partially reliable service provided by this extension is *timed reliability*. This service provides a mechanism to applications that allows them to set a time to live on packet. The time to live indicates the duration of time that the sender should try to transmit or retransmit the message before it eventually is dropped. This functionality is an extension of the *lifetime* parameter that already exist in the SCTP protocol.

3.2.3 Priority-Progress streaming

Another mechanism to prevent sending of too old data is Priority-Progress streaming (PPS) [8]. PPS combines data re-ordering and dropping to maintain timeliness of streaming in the face of unpredictable throughput. It is used to stream video. A minimal scalable compression format, derived from MPEG-1 video, called SPEG (Scalable MPEG) was developed to be used with PPS. SPEG transcodes MPEG coecients to a set of levels, one base level and three enhancement levels. Each original MPEG frame is divided into four *application level data units* (ADU), one for each level. The ADUs are classified and given priority by a Priority Mapper based on what kind of frame (I,B or P) it is and the level. The ADUs are grouped in *streaming data units* (SDU), one SDU per priority level. How the ADUs are grouped depends on how we want to scale quality. ADUs may be grouped to support frame drop or spatial drop.

The PPS algorithm is best effort. It allows the congestion control mechanism to decide appropriate sending rates. When the sending rate is low, the timeliness of the stream is maintained by dropping low-priority data, before they would otherwise reach the network. In this way, the amount of higher-priority data sent automatically matches the rate decisions of the congestion control mechanism. The PPS algorithm works by subdividing the time-line of the video into disjoint intervals called adaptation windows. PPS sends SDUs belonging to the current window. It starts by sending the SDUs with the highest priority, and continues to send SDUs with decreasing priority. If there are SDUs left when the window has passed, they are dropped. We continue at the next window with a new set of SDUs, starting by sending the SDU with the highest priority.

3.2.4 Dynamically adjusting the TCP send buffer size

In the paper *Supporting Low Latency TCP-Based Media Streams* [4] Goel, Krasic, Li and Walpole studied the feasibility of using TCP for low-latency media streaming. Their idea is to adapt the TCP send buffer size based on TCP's congestion window. The desired effect is a reduced application perceived network latency.

They investigated the protocol latency, i.e. socket to socket latency and found that a significant portion of the delay occurred due to the TCP send buffer. They also found that this delay could be eliminated by making some simple modifications to the TCP send buffer. The modifications were to dynamically adjust the send buffer size according to the current window size (CWND). A new socket option, `SO_TCP_MIN_BUF` with the parameters A and B, was introduced so the modification could be enabled on a per socket basis. The option limits the send buffer to `A*CWND+MIN(B,CWND)` at any given time. The send buffer size will always be at least CWND since A must be ≥ 1 and $B \geq 0$. They called a `SO_TCP_MIN_BUF` stream with the parameters A and B a MIN_BUF(A,B) stream.

Three different experiments with different buffer sizes were ran; MIN_BUF(1,0), MIN_BUF(1,3) and MIN_BUF(2,0). Their results showed that MIN_BUF(1,0) and MIN_BUF(1,3) streams had similar latencies and these latencies were much smaller than a MIN_BUF(2,0) stream or a regular TCP stream. While MIN_BUF(1,0) showed 30% loss in throughput, MIN_BUF(1,3) suffers less than 10%. Thus MIN_BUF(1,3) represents a good latency-throughput compromise. Their conclusion was that their experiments results showed that their simple idea reduced protocol latency and significantly improved the number of packets that could be delivered within 200 ms and 500 ms thresholds.

3.3 Summary

In this chapter, the expression LDC has been explained and the similarities and differences with partial reliability were also mentioned.

Several related work and studies on the subject have been presented and we have learned some different concepts on how to deal with the problem that packets stay too long in the send buffer and may grow old while waiting to be sent out on the wire. The DCCP API for late data choice lets an application modify the buffer content directly, thus enabling it to remove or modify packets not yet sent. The SCTP Partial Reliability Extension gives applications the ability to timestamp their messages so that they will be dropped if not sent within that time limit. Priority-Progress streaming was another mechanism described. It divides a video stream into adaption windows and give small pieces of the video different priority. By sending high-priority data first, it can drop low-priority data if it was not sent before the adaption window had passed. The last solution we looked at was to dynamically adjust the TCP send buffer size. This solution prevents an application from ever ending up in the situation where it has too many packets in the send buffer, by reducing the send buffer size while the CWND decreases.

As TCP is the most used protocol on the Internet today, it would clearly be handy to have LDC support in it. In the next chapter we will take a look at our implementation for LDC support in TCP in the Linux network architecture.

Chapter 4

Late data choice support in TCP

In this chapter, the design of the LDC support in TCP is presented. The principles and structures that makes up the system are described. Secondly, the implementation details are presented, i.e., what is modified in the kernel and how the module is implemented.

4.1 Design

The main design of TCP LDC is inspired by the DCCP API for late data choice, see section 3.2.1. The primary functionality we wanted was to be able to go back in the send buffer and delete (or modify) a packet that has not been sent yet. We also wanted to keep the existing TCP stack intact (and usable) while providing late data choice support. To achieve this, we decided to leave the existing TCP stack untouched as much as possible.

To support the ability to go back and delete (or modify) a packet that has been sent by an application, but not yet put on the wire, we decided that a send buffer in shared memory (shared between user and kernel space) would be a nice and clean solution. See figure 4.1 for a design overview.

4.1.1 The packet ring

The heart of the TCP LDC implementation is the packet ring. This packet buffer "replaces" the original packet buffer. It resides in shared memory, i.e., both the kernel and the application can access the buffer directly. This enables an application to "send" packets without issuing a system call. It just places the data directly in the buffer. When an application has put a packet in the packet ring, it notifies the kernel, if it is not already sending packets. Applications send and drop packets, notify the kernel, etc. with the aid of a LDC user space library. When the kernel receives a notification from the application, as described in section 4.2.1, it will start sending packets and continues to send as long as there are packets in the buffer that are ready to be sent. The kernel will obey the usual TCP congestion control and stop sending if the sending window is full. When the kernel stops to send for some reason, it will set a `kern_notify` flag so the application knows whether it has to notify the kernel or not, when adding

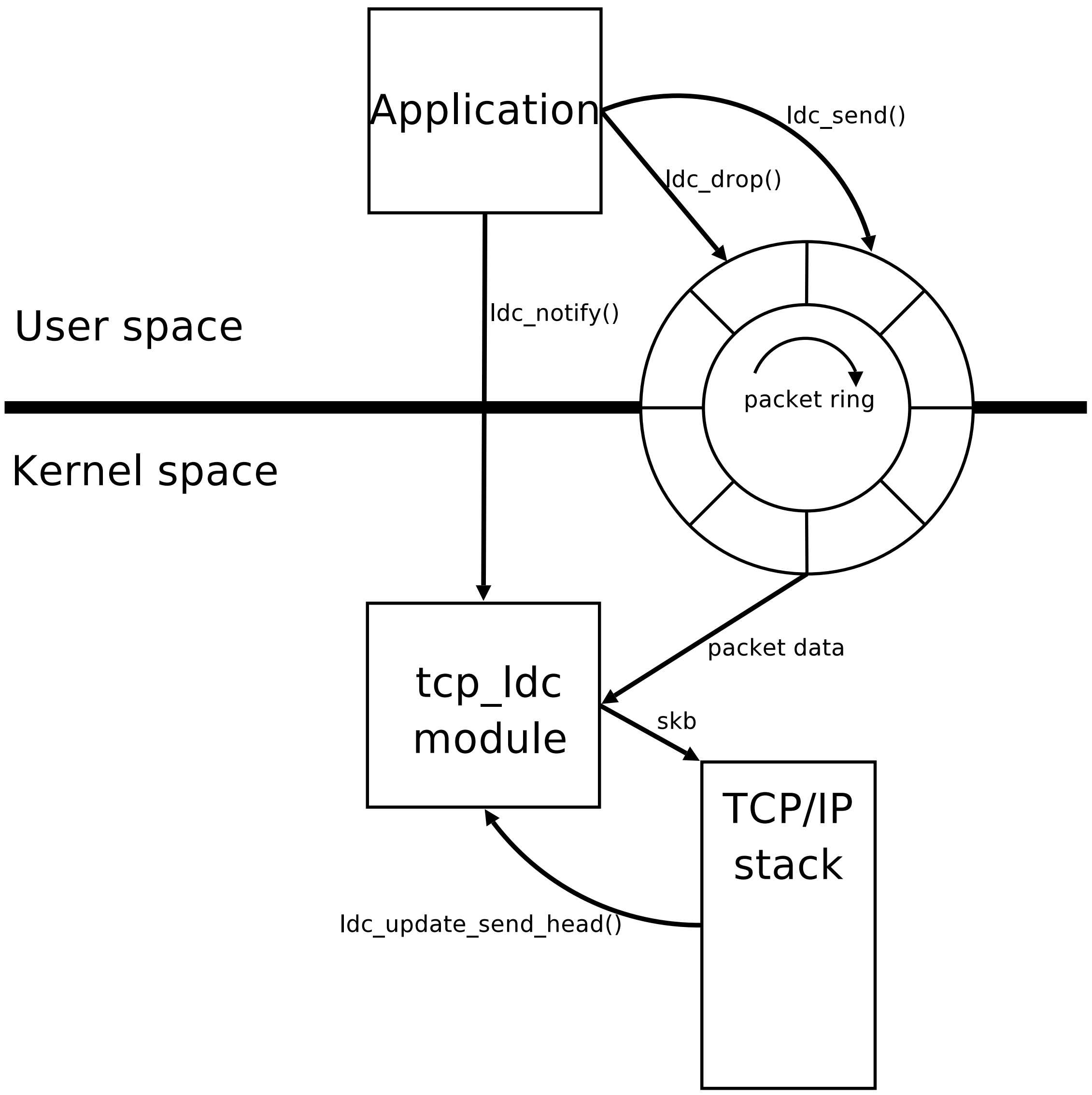


Figure 4.1: TCP LDC design

packets to the buffer.

4.1.2 Replacing the existing socket buffer

As TCP is a fairly complex protocol, a new buffer would have to support all the features of TCP (like congestion avoidance and retransmit). It would lead to an unnecessary complex design (and redundant implementation) to actually replace the existing buffer. Also, much of the output part of the TCP stack would have to be rewritten/extended to support a new buffer as it depend on the existing. Because of that, the new buffer would be placed on top of the existing buffer. To prevent the LDC buffer from leading to even more delay and not getting the ability to delete or modify the packets, the TCP LDC buffer is not just put on-top of the existing buffer, it is integrated. With TCP LDC all unsent packets will be in the TCP LDC buffer, making it possible to modify/delete packets. Only one packet, the next to be sent, will be in the exist-

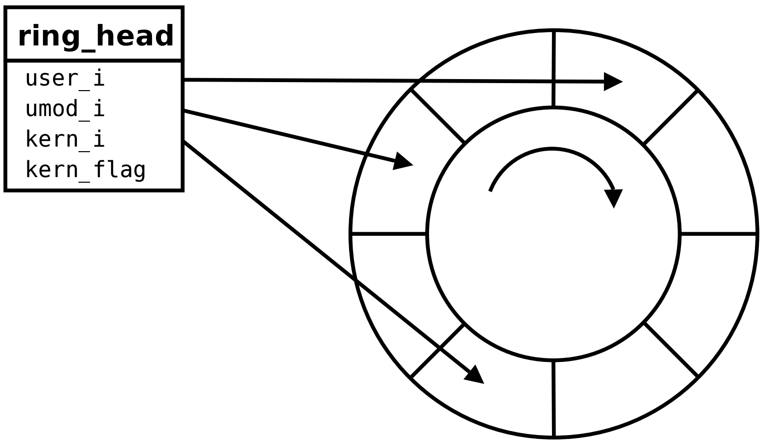


Figure 4.2: TCP LDC packet ring and indexes

ing send buffer. Packets in the TCP LDC buffer is just data, with a minimal TCP LDC header, and packets in the original send buffer is `skb`s. Thus, packets that are moved to the original buffer is wrapped as `skb`s on the fly. This will be explained in greater detail in section 4.2.

4.1.3 Indexes and flags

There are three indexes used by the packet ring.

- `kern_i` - the kernel index. Only updated by the kernel.
- `umod_i` - the user modification. Only updated by the application.
- `user_i` - the user index. Only updated by the application.

Slots with index `i` where `kern_i` $\leq$ `i < umod_i` are packets ready to send. Slots with index `i` where `umod_i` $\leq$ `i < user_i` are packets that the application freely can modify/delete. The next slot where the application can add a packet is at index `user_i`. Slots with index `i` where `user_i` $\leq$ `i < kern_i` are free slots/sent packets. During normal operation, `umod_i` equals `user_i`. `umod_i` is only decremented (to the index where the target packet is), when the application wants to modify or delete a packet, in order to hold back the kernel. See figure 4.2 for a graphical representation of the TCP LDC packet ring and indexes.

There are two types of flags; `kernel_flag` and `user_flag`. Is is only one `kernel_flag` and it is in the structure that describes the packet ring, `struct packet_head`. If `kernel_flag` is set, the kernel is not sending packets and needs to be notified to start sending. Every packet in the packet ring has a `user_flag`. If the `user_flag` is set, the packet is tagged as deleted, and the kernel will skip it while sending. This is very similar to how it is done in the DCCP API for LDC, shown in figure 3.4.

4.1.4 Security

In this section, we will discuss some security issues that one may need to address to make TCP LDC more robust. In principle, an application can only hurt itself by setting invalid values in indexes etc. However, a robust kernel implementation should not let the users make (at least stupid and obvious) errors. A careless application may for instance, for some unknown reason, modify the `kern_i` index. To prevent undefined situations, the kernel may want to keep its own copy of that index. The kernel may also want to keep copies of the user indexes, to check that a change looks sane. At least, a basic sanity check, e.g., assuring that `kern_i` $\leq$ `umod_i` $\leq$ `user_i` is always true, should be implemented.

4.2 Implementation of TCP LDC in the Linux kernel

In this section, the actual implementation of TCP LDC is explained. The implementation of LDC support in TCP consist of the following: A module, various changes to the kernel to interface with the module and a user space library.

4.2.1 Modifications of the kernel

To make TCP LDC work, the kernel itself has been modified in several places. In most cases, it is just an `EXPORT_SYMBOL` to make functions and variables needed by the TCP LDC module available. Other changes are the implementation of `TCP_LDC` option in `setsockopt()` and `getsockopt()`. The last thing that is implemented outside the module is the notification interface. See table 4.1 for a list of functions mentioned in this section and their source files.

setsockopt()

To be able to define a socket as a TCP LDC socket, a new option, `TCP_LDC`, was defined on the TCP level (`SOL_TCP`). The option values for the `TCP_LDC` option are the buffer size, in number of packets, and the packet size, in bytes. The kernel will try to allocate a buffer, sized *packets* * *packetsize* (+ *meta-data*), for the application. Be aware that `setsockopt()` may fail setting the `TCP_LDC` option if there are not enough free memory. If `tcp_setsockopt()` gets a `TCP_LDC`, it does three things:

- Loads the `tcp_ldc` module if not already loaded (see section 4.2.2 for details on the module).
- Sets the `SOCK_LDC` flag on the socket to indicate that this socket is now a LDC capable socket.
- Calls `tcp_set_ldc()` in the `tcp_ldc` module with the option value as parameter.

Function	**Source file**
`__tcp_push_pending_frames()`	net/ipv4/tcp_output.c, line 1079
`get_unmapped_area()`	mm/mmap.c, line 1331
`insert_vm_struct()`	mm/mmap.c, line 1963
`kmem_cache_alloc()`	mm/slab.c, line 2803
`memset()`	lib/string.c, line 447
`skb_add_data()`	include/linux/skbuff.h, line 1119
`skb_copy_to_page()`	include/net/sock.h, line 1056
`tcp_getsockopt()`	net/ipv4/tcp.c, line 1934
`tcp_ioctl()`	net/ipv4/tcp.c, line 404
`tcp_push_pending_frames()`	include/net/tcp.h, line 878
`tcp_sendmsg()`	net/ipv4/tcp.c:662
`tcp_setsockopt()`	net/ipv4/tcp.c, line 1690
`tcp_write_xmit()`	net/ipv4/tcp_output.c, line 999
`update_send_head()`	net/ipv4/tcp_output.c, line 54
`vmalloc_32()`	mm/vmalloc.c, line 554

All line numbers refers to a unchanged 2.6.15.4 source tree.

Table 4.1: Functions mentioned in section 4.2 and their source files

getsockopt()

In `getsockopt()`, the same option (at the same level) as in `setsockopt()` is used. When `tcp_getsockopt()` gets a `TCP_LDC`, it puts the *user space* address to the shared buffer (packet ring) in parameter for the return value, given by the user. Now, the application has access to the packet ring and can start placing packets in it.

Notifications

When the application adds packets to the packet ring (and kernel is not currently sending packets) it has to tell the kernel that there now is work to do. To notify the kernel, a special `ioctl()` request for TCP LDC sockets, `SIOCTCPLDCNTF`, was implemented. If the `kern_notify` flag is set when the application is adding a packet to the packet ring, it should notify the kernel. Note that the application does not have to notify the kernel even if the `kern_notify` flag is set. It may choose to fill the packet ring a bit (or even completely full) before letting the kernel loose on the ring. When `tcp_ioctl()` receives a `SIOCTCPLDCNTF` request, it calls `ldc_notify()` in the `tcp_ldc` module.

4.2.2 The TCP LDC kernel module

The TCP LDC module, named `tcp_ldc` (net/ipv4/tcp_ldc.c), is the core implementation of TCP LDC. It implements buffer initialization and handling, sending packets and removing deleted packets on behalf of the application.

The `tcp_ldc` module has these main functions, which are described in the next paragraphs.

int tcp_set_ldc(struct sock *sk, char __user *optval, int optlen)

This function is called when an application defines its socket as an LDC socket (with the `setsockopt()` call). When the application want to LDC enable its socket, it passes along a pointer to a variable with the desired packet ring size, in number of packets and the packet size. After the variable value is copied to kernel space, we allocate some memory to fit a `struct ldc_packet_ring` (see listing 4.1) at `sk->ldc_ring`. As we now have a pointer to the packet ring in the socket, we are able to reach the packet ring every time we get a call in the module. A pointer to the socket, `sk`, is always included as one of the parameters in calls to functions in the module. After the socket has its LDC data in place, we call `ldc_pkt_buffer_alloc()` to do the actual work with allocating and mapping the packet ring. When we are finished allocating the actual packet ring, the module will "sleep" until it gets a notification from the application. See appendix A.1 for the full source code for this function.

```
struct ldc_packet_ring {
    struct ldc_ring_head *ring_head;
    struct ldc_packet    *packet_ring;

    /* Buffer addr in user space */
    void                 *ldc_pkt_buffer_uaddr;
    /* Buffer addr in kernel space */
    void                 *ldc_pkt_buffer;
    /* Aligned size */
    int                   ldc_pkt_size;
}
```

Listing 4.1: The ldc_packet_ring struct

int ldc_pkt_buffer_alloc(struct ldc_packet_ring *lpr, int rsize, int psize)

This function allocates and shares the packet ring between the kernel and the application. It receives the requested ring size, given in packets, as a parameter. The first thing this function does is to calculate the size of the ring i bytes:

```
buffer_size = sizeof(struct ldc_ring_head)
    + (rsize * sizeof(struct ldc_packet))
    + (rsize * psize);
size = PAGE_ALIGN(buffer_size);
```

As you can see, the size is page aligned. This means that the size is rounded up to the next page boundary. The reason for doing this is that memory is mapped at a per page basis, and to assure that no one else will interfere with our ring, we make sure we use whole pages. Now, it is time to allocate some memory. This is done by the helper function `ldc_rvmalloc()`. When the memory is allocated we call `kmem_cache_alloc()` to grab a virtual memory area cache, `vma`. The `vma` will be used to describe the allocated memory (the packet ring) and later merged with the application's virtual memory. Now, we find some free space

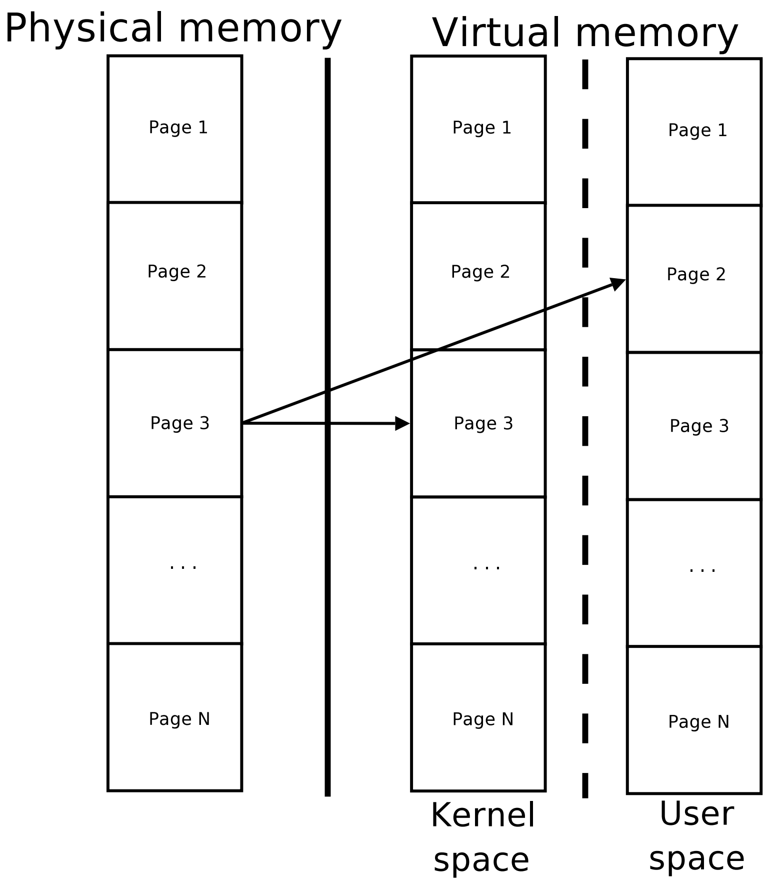


Figure 4.3: Mapping memory into the application's memory space

with the size of the (page aligned) packet ring in the application's address space (`get_unmapped_area()`). Then, we map the packet ring in the `vma`, which is done by the helper function `ldc_remap_buffer()`. The next thing to do is to initialize the packets, see listing 4.2. This is done by calculating the kernel- and user-space address, and set the values in the `ldc_packet` structs. Then, we merge our new `vma` with the application's address space, see figure 4.3, this is done with a call to `insert_vm_struct()`. The user space address is stored in the socket so it can be retrieved with `getsockopt()`. Finally, we initialize the ring with some default values. Setting all indexes to zero and setting the `kern_flag`.[1] See appendix A.2 for the full source code for this function.

```
struct ldc_packet {
    int user_seq;
    unsigned int kern_flag : 2;
    unsigned int user_flag : 2;
    int size;
    char *data;  /* Pointer to the data in USER SPACE */
    char *kdata; /* Pointer to the data in KERNEL SPACE */
    int offset;
```

[1]See section 4.2.4 for some interesting issues encountered while implementing this function.

```
}
```

Listing 4.2: The ldc_packet struct

int ldc_notify(struct sock *sk)

Since the packet ring is shared between the application and the kernel, we do not use the normal write/send etc system-calls to send packets, but we just put them in the packet ring (with some help of the LDC user space library). To start sending packets, we need to notify the kernel to tell it that packets now exist in the buffer. We need to notify the kernel when the `kern_flag` is set (this is set by the module on initialization of an LDC socket, and every time the buffer get empty). The kernel is notified by the user space call `ldc_notify()` (see section 4.2.3). The notification itself is implemented as an `ioctl()` call. To notify with `ioctl()`, you issue a request `SIOCTCPLDCNTF` (0x894b) on the socket's `fd`. When the kernel receives a `SIOCTCPLDCNTF`, it calls `ldc_notify()` in the `tcp_ldc` module. `ldc_notify()` is a very simple function that clears the `kern_flag` and calls `ldc_sendmsg()` to start sending packets. See appendix A.3 for the full source code for this function.

int ldc_sendmsg(struct sock *sk)

`ldc_sendmsg()` is a slightly modified `tcp_sendmsg()`. Instead of taking what to send as a parameter (like `tcp_sendmsg()` does), `ldc_sendmsg()` sends the next packet from the LDC packet ring. It uses `ldc_next_pkt()` to get the index in the packet ring of the next packet to send. Since the buffer is shared, and it has a kernel space address, the helper functions, `ldc_skb_add_data()` and `ldc_skb_copy_to_page()`, are used to add data to `skb`s.

As with `tcp_sendmsg()`, `ldc_sendmsg()` makes a `skb` and inserts it into the sockets `sk_write_queue` before calling `tcp_push_pending_frames()` etc. This behavior is kept (as discussed in section 4.1.2) to support retransmissions without altering that part of the TCP code. If the packet in the LDC packet ring is larger than the size allowed for the `skb` at the moment, the packet is split up and copied into several `skb`s. The allowed size of the `skb`s are determined by the current maximum segment size, `mss_now`.

int ldc_next_pkt(struct sock *sk)

`ldc_next_pkt()` is used to determine the index of the next packet to send. It uses `kern_i` as the starting point, as this is the index to the next packet to send. The reason for having the `ldc_next_pkt()` function is that there are two conditions where you can not simply return the `kern_i` value. These two are when the buffer is empty and when a packet is deleted. The very first action taken is to check whether the buffer is empty or not. If not, we check whether the packet at `kern_i` has the `user_flag` set or not. If not, we return `kern_i`. If `user_flag` were in fact set, then the packets is ignored and `kern_i` is incremented, by one, and we start all over. If the buffer is not empty after removing any deleted packets, `kern_i` is returned. If the ring is empty, the notification flag, `kern_flag`, is set and `-1` i returned. See appendix A.4 for the full source code for this function.

struct sk_buff *ldc_update_send_head(struct sock *sk, struct tcp_sock *tp, struct sk_buff *skb)

As discussed in section 4.1.2, the existing socket buffer is not 100% replaced. Only the part holding unsent packets is replaced. We still use the original buffer to do retransmit etc. To accomplish this, `update_send_head()` calls `ldc_update_send_head()` for LDC sockets when the send buffer is empty. As described in section 2.5, `update_send_head()` is called in `tcp_write_xmit()`. `tcp_write_xmit()` is called when new packets are to be sent and when the sending window grows (both cases via `__tcp_push_pending_frames()`). The original `update_send_head()` has the trivial task of just fetching the next `skb` in queue. The problem for LDC sockets is that there are only packets in the send buffer when a packet from the ring buffer was just wrapped as one or more `skb`s. Our unsent packets are in the packet ring. To address this, `ldc_update_send_head()` is called for LDC sockets if the send buffer is empty, see listing 4.3. See appendix A.5 for the full source code for this function.

```
static inline void update_send_head(struct sock *sk, struct
   tcp_sock *tp, struct sk_buff *skb)
{
     sk->sk_send_head = skb->next;
     if (sk->sk_send_head == (struct sk_buff *)&sk->
        sk_write_queue)
          if(sock_flag(sk, SOCK_LDC))
               sk->sk_send_head = (*ldc_update_send_head)(sk, tp,
                    skb);
          else
               sk->sk_send_head = NULL;
     tp->snd_nxt = TCP_SKB_CB(skb)->end_seq;
     tcp_packets_out_inc(sk, tp, skb);
}
```

Listing 4.3: Modified update_send_head()

`ldc_update_send_head()` takes the next (if any) packet from the ldc packet ring (calling `ldc_next_pkt()`), puts it in one or more (new) `skb`s, entails them in the socket's send buffer, and sets `sk->sk_send_head` to the new skb, or `NULL` if there are no packets to send (empty buffer). The actual work of allocating new `skb`s and entailing them in the send buffer, is done by the function `ldc_make_new_skb()`. The index of the packet in the LDC packet ring to entail in the send queue, is passed as a parameter.

struct sk_buff *ldc_make_new_skb(struct sock *sk, int idx)

As the work that has to be done by `ldc_make_new_skb()` is pretty similar to what `ldc_sendmsg()` does, it also is based on `tcp_sendmsg()`. `ldc_make_new_skb()`

is a more modified version of `tcp_sendmsg()` than `ldc_sendmsg()`. We do not call `tcp_push_pending_frames()`, and etc., since we already are in a sending state. So, instead of starting a transmission, we return a pointer to the new `skb` (or `NULL` if we failed for some reason). We also use the helper functions `ldc_skb_add_data()` and `ldc_skb_copy_to_page()` for the same reasons as in `ldc_sendmsg()`. If the packet in the LDC packet ring is larger than `mss_now`, the packet will be split into several `skb`s. All the `skb`s will be entailed in the send buffer. In those cases, `ldc_make_new_skb()` will return the pointer to the first `skb` it entailed in the send buffer. `update_send_head()` will take care of all the `skb`s in the send buffer before calling `ldc_update_send_head()` (and with that `ldc_make_new_skb()`) to entail more `skb`s.

Helper functions

- **`ldc_rvmalloc()`** - Allocates memory for the packet ring. Assures that the size is page aligned, uses `vmalloc_32()` to allocate memory and clears it with `memset()`.
- **`ldc_rvfree()`** - Frees packet ring memory. Only a wrapper for `vfree()`.
- **`ldc_remap_buffer()`** - Maps the packet ring in a `vma`. Uses `vmalloc_to_page()` for all page aligned addresses in the allocated memory to get a `struct page`. Uses `vm_insert_page()` on each page to map them in the `vma`.
- **`ldc_skb_add_data()`** - Adds data to an `skb`. It is a modified `skb_add_data()` to enable the from address to be a kernel space address.
- **`ldc_skb_copy_to_page()`** - Adds data to an `skb`. It is a modified `skb_copy_to_page()` to enable the from address to be a kernel space address.

4.2.3 The TCP LDC user space library

A user space library was implemented to assist the application programmer. The functions provided by the library helps the programmer to set up a LDC socket, send packets via LDC and drop packets.

struct ldc_packet_ring *ldc_init(int socket, int ring_size, int packet_size)

This functions initializes the socket as a LDC socket. It receives the socket, the desired ring size in packets, and the packet size as parameters. `ldc_init()` starts by allocating memory for the `ldc_packet_ring` struct. Then, it uses `setsockopt()`, with the `TCP_LDC` option, to tell the kernel about the desired ring and packet size. Thereafter, it uses `getsockopt()` to get the address to the packet ring. The pointers to `ring_head` and `packet_ring` are set in `ldc_packet_ring`, and the pointer to the struct is returned. If `setsockopt()` or `getsockopt()` fails, the memory allocated is freed and `NULL` is returned.

int ldc_send(struct ldc_packet_ring *lpr, int user_seq, void *addr, int size, int notify)

The `ldc_send()` function “sends” packets, i.e., adding packets to the ring buffer. It uses `addr` and `size` to copy the data into the packet ring at index `user_i`, if the buffer is not full. If `size` is larger than the packet size requested in `ldc_init()`, several packets are added to the ring. Note that all packets added to the packet ring in one `ldc_send()` call, will get the same `user_seq`. For each packet added to the ring, `user_i` is incremented by one, and `umod_i` is set to the same value as `user_i`. If `notify` is non-zero and the `kern_flag` is set, the kernel will be notified (with `ioctl()`). `ldc_send()` returns the amount of data “sent”, i.e., added to the packet ring.

int ldc_drop(struct ldc_packet_ring *lpr, int user_seq)

`ldc_drop()` searches the packet ring for one or more packets with sequence number `user_seq`, and mark them as deleted by setting the `user_flag`. The number of marked packets are returned. Note that one or more of the packets marked, may already been sent.

4.2.4 Implementation choices and issues

In this section, we discuss some choices made and issues encountered while implementing the LDC support for TCP.

How to notify?

An alternative method for kernel notification was implemented. The alternative hijacks any calls ending up in `tcp_sendmsg()` (like `write()` and `send()`). It does not matter what you try to send (it would just be ignored). The kernel would check if the `fd` used is a TCP LDC socket, if it is, it will only call `ldc_notify()` and return (just as the `ioctl()` way of notifying the kernel). This alternative was developed for debugging purposes. At some point in the development, it was desirable to try a new way of notifying. This notification mechanism still exist in the implementation of TCP LDC. It may be nice to have this call for compatibility reasons. If one used the regular `write()` or `send()` calls, the data would end up in the send buffer (and not the packet ring) and would be prioritized over the data in the ring buffer, due to the way `update_send_head()` works.

The packet ring - a problem area

The shared buffer was first implemented as just a meta-data ring, the actual packet data was not shared. Packet data could be anywhere in the applications user space memory, and only the address and size were added to the packet ring. It was the application that allocated the shared memory (page aligned, only one page in size) and passed the address to the kernel with the `setsockopt()` call. The kernel would map the address given from the application into its address space. The functions `ldc_sendmsg()` and `ldc_make_new_skb()` used the regular `skb_add_data()`

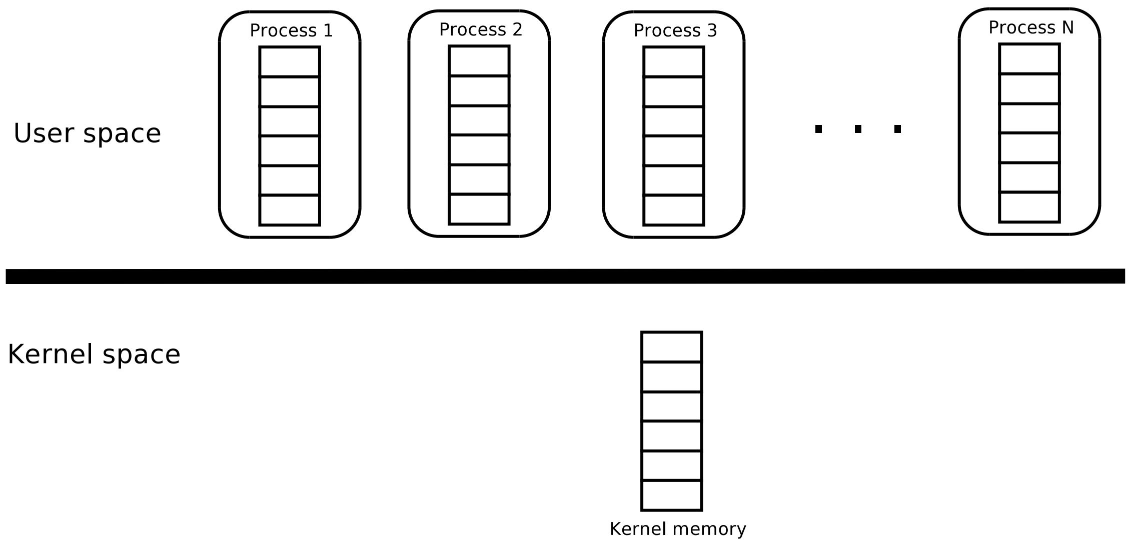


Figure 4.4: User space memory in kernel context

and `skb_copy_to_page()` calls. `skb_add_data()` and `skb_copy_to_page()` use the regular `copy_from_user()` calls to copy data, that fitted nicely with the rest of the design. The implementation seemed to work, but only almost. Sometimes the functions that copied data from user space returned `BAD ADDRESS`. After some debugging and investigation, the reason became pretty obvious. When we received an ACK from the other side of the connection, `ldc_update_send_head()` would be called if the sending window opened enough to send a packet. Here we found the problem; when `ldc_update_send_head()` called the copy functions, they would return `BAD ADDRESS` because the kernel was in an interrupt, and thus in kernel context. In that context, there is no reference to the user space memory of the application owning the socket. As we do not have that necessary reference, we do not know which user space to copy from, see figure 4.4. So, when the address and size was fetched from the packet ring and passed to the copy functions, the functions did not know what to do. As a result, the packet ring implementation was redesigned to the solution used now, described in section 4.2.2.

Performance

While doing some initial testing, the implementation performed very poor. The highest rates achieved, were approximately 250 kbit/s. The initial implementation had a packet size-limit of 1460 bytes (based on a 1500 bytes MTU and 40 byte headers). The user space function `ldc_send()` always notified the kernel when the `kern_flag` was set. It seemed that the performance was poor due to the fact that the application never got ahead of the kernel. The application would add a packet and notify the kernel right away, and the kernel sent the packet and sat the `kern_flag`. Thus, when the application added the next packet, it had to notify the kernel again. This send-notify loop seemed to hold back the send rate. Therefore, these two sections were modified in an attempt to increase the performance. The packet size-limit was removed, and instead the user selects the maximum packet size while initializing the LDC socket with `ldc_init()`. The function `ldc_send()` got a new parameter, `notify`, that tells the function if it should notify the kernel or not. This enables the application to fill the buffer before sending. By filling the buffer, we get a head-start on the kernel and may reduce the need for notifications. When the modifications were done, new tests were run. Now it was able to send at rates above 3 Gbit/s, a distinct improvement.

4.3 Summary

In this chapter, the design of the LDC support in TCP was presented. The principles and structures that makes the system were described, and the implementation details were presented. We saw what is modified in the kernel and how the module is implemented. We also took a look at some implementation choices and issues. In the next chapter, an evaluation of TCP with and without LDC support is given.

Chapter 5

Testing

In this chapter, we evaluate the LDC support for TCP. Test results are compared and discussed. The goal was to find out if we with LDC support could send only relevant data, within a given time limit, and discard old data from the send buffer. By only sending relevant data we assure that the send buffer do not contain outdated data, and this makes it easier to hold deadlines. It was not the intention to find out how much resources, CPU cycles, memory etc., the LDC support for TCP uses compared to regular TCP with these tests.

5.1 Test setup

The evaluations were carried out on a test setup with three computers; one sender, one network emulator and one receiver. See figure 5.1 for a visualization of the test setup. NetEm [6] was used on the middle machine as network emulator. Two programs were written to run the tests, one sender and one receiver. The sender simulates a media server that streams data with a rate of 1 Mbit/s divided into 8 layers of 128 kbit/s (1 Mbit = 1024 kbit = 2^{20} bit). The first layer of the stream is the base layer. To play any content at all, the receiver needs a complete base layer. The other 7 layers are enhancement layers. Layer 2 (enhancement layer 1) extends layer 1 (the base layer), layer 3 (enhancement layer 2) extends layer 2 and so on, see figure 5.2. The more layers the receiver gets, the better the quality of the media will be. As the layers are dependent on each other, they have to be received in the correct order. E.g., if you received layer 1, 2, 3 and 5, you can not use layer 5 since you are missing layer 4. The layers must also be received in a timely manner. If they are received too late, they can not be used. The sender application sends one layer of 128 kbit every 1/8 second, i.e. a total of 1 Mbit/s. The receiver is a very simple application. It receives data from the sender for 60 seconds, and print some statistics. When the receiver has finished, it outputs how much data it got, the average receiving rate, and the layer distribution. The layer distribution show how much data the receiver got on each layer.

Four tests were run, with different network emulator configurations, both with and without LDC support. The four network emulator configurations were:

- No rate limit, no loss.
- 1024 kbit/s, 5% loss.

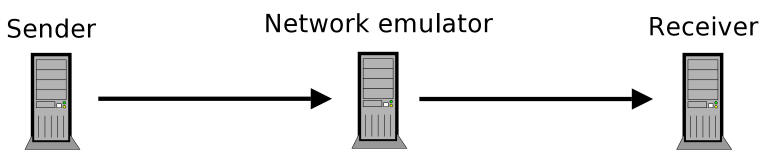


Figure 5.1: Test setup

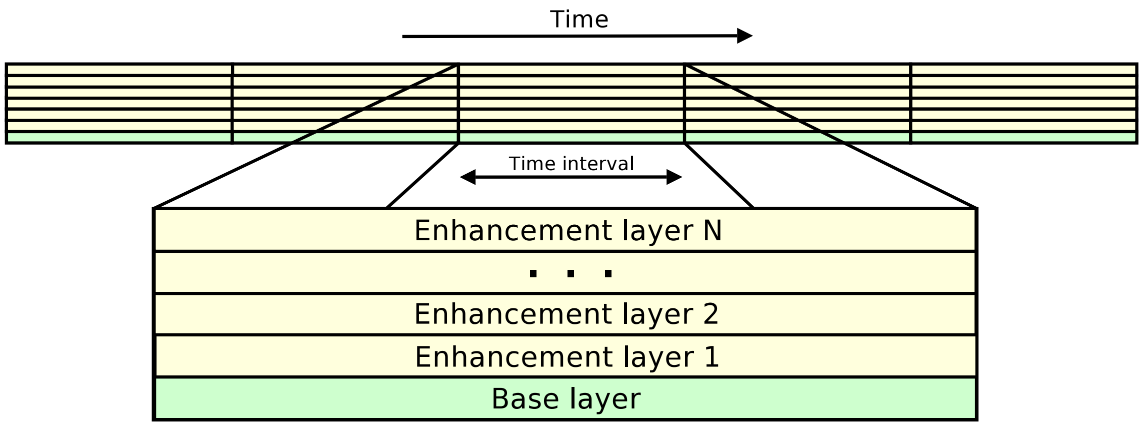


Figure 5.2: Layered video

- 512 kbit/s, 5% loss.
- 256 kbit/s, 5% loss.

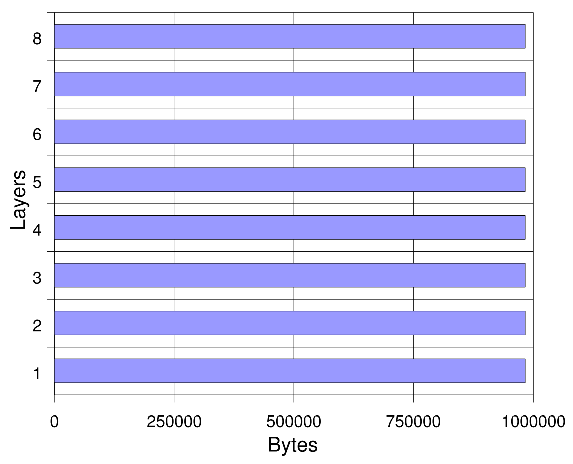
Some may find a loss percentage of 5 to be artificially high, but this is done on purpose to provoke delays due to retransmissions.

5.1.1 TCP setup

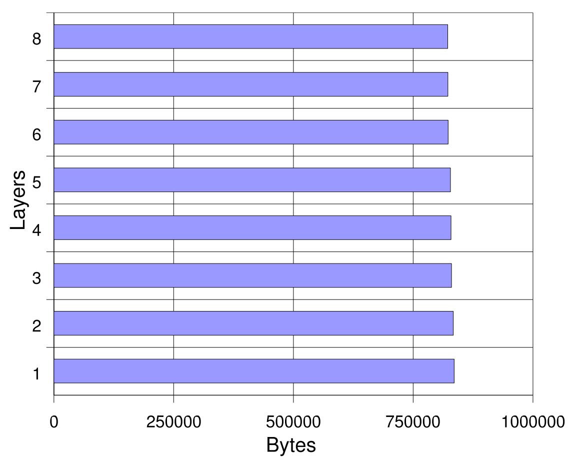
The test setup for TCP without LDC support, i.e. regular TCP, consisted of the sender application, the network emulator and the received application as described. In Linux, the NewReno [2] TCP variant is the default. The sender application has two modes. In the *no-ldc* mode, it uses the regular `send()` system call instead of `ldc_send()`.

5.1.2 TCP LDC setup

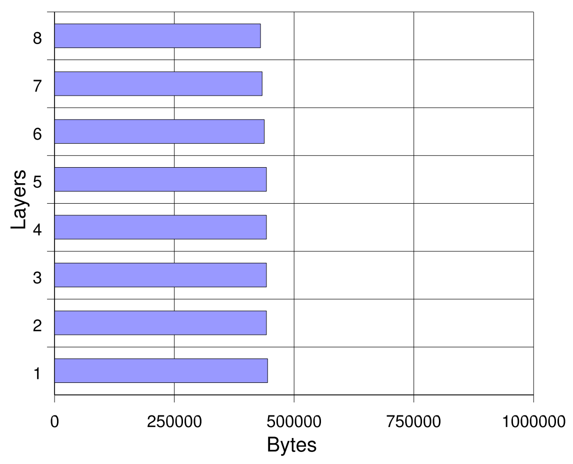
The same three parts of the test setup already mentioned, were used in the evaluation of TCP with LDC support. The only difference was the sender mode. As the regular TCP sender mode used the `send()` system call, the TCP LDC sender mode used the `ldc_send()` and `ldc_drop()` from the TCP LDC user space library described in section 4.2.3. In TCP LDC mode, the sender application would do some additional tasks to just sending data. After it had sent one time window of data, 1 second in this application, it would check if we still had packets in the packet ring. If we had, they would be dropped, starting at the least significant enhancement layer, and drop



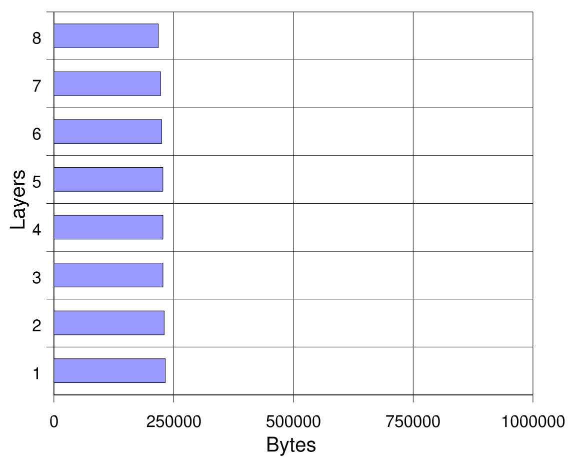
(a) No rate limit, no loss



(b) 1024 kbit/s, 5% loss



(c) 512 kbit/s, 5% loss



(d) 256 kbit/s, 5% loss

Figure 5.3: TCP test results

as many layers as needed in a descending order. When done, we would start on a new time window by sending the base layer.

5.2 Results

In this section the results from the evaluation runs are presented. The results from the tests with regular TCP are presented first, and then we take a look at the results from the TCP with LDC support tests.

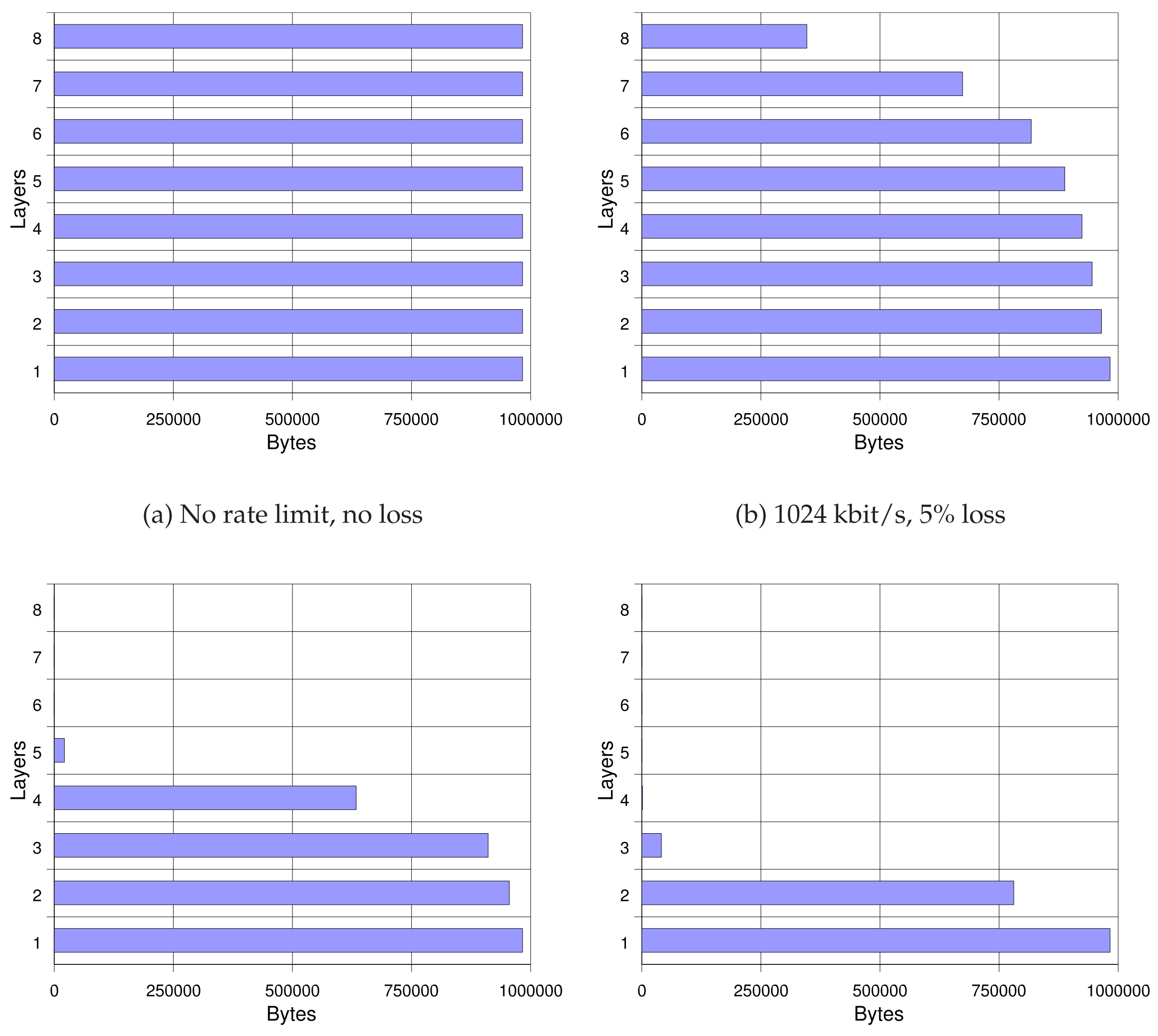
5.2.1 TCP results

The tests were run ten times and the values presented here is the average of these ten runs. As described in section 5.1, we had four configurations. In figure 5.3 you can see the results we got with regular TCP. The first test was with no rate limit and no loss, shown in figure 5.3(a). Here we received 983040 bytes for each layer, a total of 7.5 MB. 983040 bytes = 128 kbit/s * 60 sec. This tells us that we got the entire stream of 1 Mbit/s the whole 60 seconds. When we in the next test run introduced a rate limit of 1024 kbit/s and 5% loss, shown in figure 5.3(b), we got a total of 6.32 MB, evenly distributed on the eight layers. The reason for the even distribution is that we do not drop any packets. Since TCP is a reliable protocol, we get all the data from all the layers we send. If you take a close look at the figure, you may see that the data is not entirely even distributed, the least significant layers have just a bit less data. This is because we send one second of data from one layer at the time, starting at the base layer. When the 60 seconds has passed, it is random where in the send process we are, but we know that we always sent the most significant layers first. Figure 5.3(c) show the results from the tests with 512 kbit/s rate limit and 5% loss. We got a total of 3.35 MB with this configuration, also evenly distributed between the layers. The last test, with a rate limit of 256 kbit/s and 5% loss, shown in figure 5.3(a), gave a total of 1.73 MB data transferred.

In a streaming point of view, only the test with no rate limit and no loss is good enough. We can only play back the length of the received base layer. In a situation with real-time streaming, we would not be able to play back the media at all. This is because we after just a short amount of time would start to receive the base layer data too late. With TCP we would get the full quality of the whole stream, since we have all the layers. Unfortunately, we will have no use of that quality when the data is received at a too slow rate.

5.2.2 TCP LDC results

The evaluation of TCP with LDC support used the same four configurations as for regular TCP, and also here the values presented are an average of ten runs. Test results for TCP with LDC is presented in figure 5.4. The results for the no limitation test, shown in figure 5.4(a), are identical to the regular TCP ones. In the next test, with 1024 kbit/s rate limit and 5% loss, the LDC effect is shown, see figure 5.4(b). In this test we got 6.24 MB, a bit less than with regular TCP. Possible reasons for this are discussed in section 5.2.3. Although we got a little less data in the same time period than with regular TCP, the data we observe at the receiver is much more relevant. The base layer, layer 1, is full, meaning that we can present the media at the receiver in the correct time-frame. The figure clearly illustrate the effect we wanted to achieve, which is that the most significant layers are prioritized and the less significant layers are given less priority. This effect is a result of dropping packets still in the send buffer, after the time window has passed. The two last test configurations, 512 kbit/s and 256 kbit/s with 5% loss, shown in figures 5.4(c) and 5.4(d), also show the same effect. The base layer is full and the amount of data received on every layer gradually decrease as they become less significant. The amount of data received in the two last tests were 3.34 MB and



(a) No rate limit, no loss

(b) 1024 kbit/s, 5% loss

(c) 512 kbit/s, 5% loss

(d) 256 kbit/s, 5% loss

Figure 5.4: TCP LDC test results

	Regular TCP	TCP with LDC	Difference
1024 kbit/s, 5% loss	862.47 kbit/s	851.96 kbit/s	1.22%
512 kbit/s, 5% loss	457.72 kbit/s	456.30 kbit/s	0.31%
256 kbit/s, 5% loss	235.88 kbit/s	235.24 kbit/s	0.27%

Table 5.1: Receive rate

1.72 MB.

5.2.3 Comparison

The results show that we can deliver a lot more relevant data with LDC, than with regular TCP. As already mentioned in section 5.2.1, the data delivered by regular TCP, in the settings with limitations, is not very usable. If we were to implement a layered streaming architecture with regular TCP, we would have to check available resources before sending a layer, and not after as our test application does. So a direct comparison between the results may not be completely fair. Nevertheless, the test results clearly states that streaming of layer encoded media greatly benefits from LDC support. Let us also take a look at the performance, the measured throughput. We mentioned in the previous two sections how much data the test application received in each configurations. We discovered that we with LDC received a bit less data than without LDC. If we convert the amount of data to throughput, we get the values shown in table 5.1. As we can see, TCP with LDC appears to be a bit slower than regular TCP. We have not done any investigation on why we get these results, but we have discussed some possible 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 the fact that we are actually dropping packets. In our test applications, we have a LDC packet size equal to data for one layer in one second, i.e., 128 kbit. So when we are dropping packets, we may not fully utilize the available bandwidth. This may be tuned with different LDC packet sizes. The important thing to remember is that even if we have a bit lower throughput with LDC support compared with regular TCP, all the data received with LDC support is usable in contrast to the data received with regular TCP. A good throughput is useless if the data transmitted is already too old.

5.3 Discussion

As shown by the test results presented, TCP with LDC support does a great job providing mechanisms to the sender application that enables it to only send relevant data. This is very nice for the receiver, since all the data it get is relevant and can be used. Since the LDC support it self has no logic for dropping packets, this has to be handled by the application. This is very flexible, but the drawback is that the application must have a good drop algorithm to have useful results. Improper use of the LDC support may lead to unwanted results, like the results in figure 5.5 show. As you can see, we got very little data in the middle layers when the bandwidth is limited. This is clearly bad, since we can not use data from a higher level if we do not have data from the underlying layers, in that time window. In this particular example the time window is equal to 1/8 sec, one layer, and we drop packets waiting in the packet ring. By doing this we risk dropping the layer just sent, e.g layer 4, and then make room for layer 5. So to make LDC useful, the application programmer must know what she does.

As the test results show, applications that send data with a limited lifetime, like media and game servers, 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

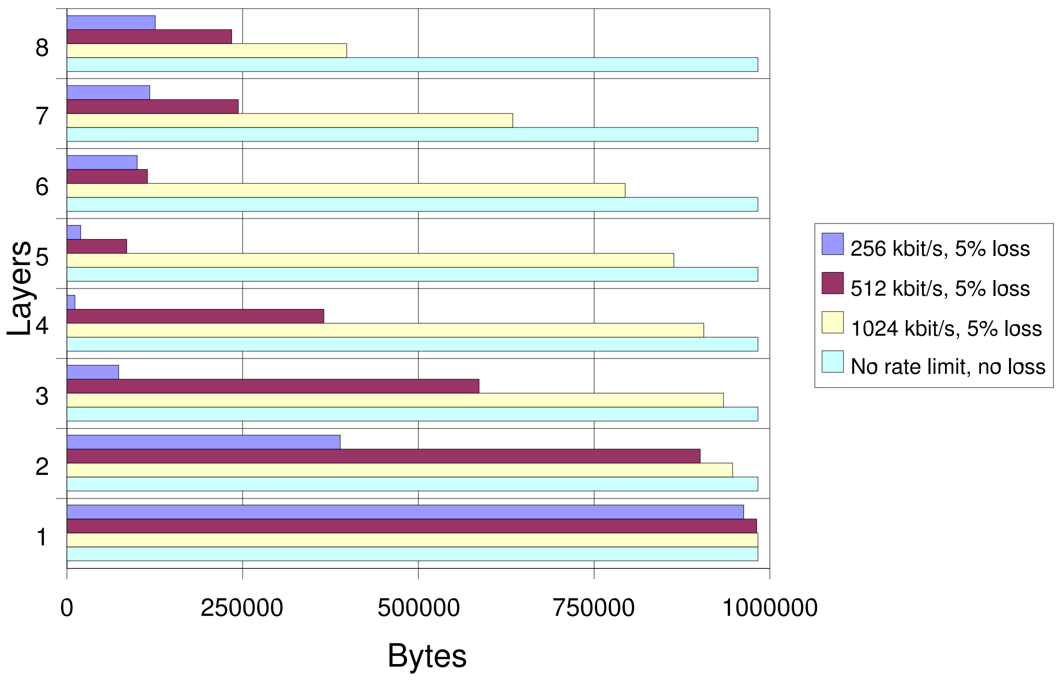


Figure 5.5: Results from an application that used LDC improperly

socket as a 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 discussed further in section 6.3, 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.

Please note that the test results, presented in section 5.2, does not say anything about timing and timeliness on the delivery of packets. This means that some of the packets may have been delivered to late because of delays in the network or because it was lost and retransmitted. However, we observed that the whole base layer was delivered within the time of the test run when LDC was used. This indicates that only minor delays may have occurred.

5.4 Summary

In this chapter, TCP with and without LDC support was evaluated. The test results were compared and discussed. We found that we can deliver relevant data, and drop useless data from the packet ring with LDC support in TCP. This enables the receiver to playback the media in a timely manner and only reduce the quality if not all layers are delivered. The next chapter presents the main conclusion. Ideas for further work with LDC support for TCP are also presented.

Chapter 6

Conclusion and further work

In this chapter, we first summarize what was done to support LDC in TCP and how it was done. Then, a conclusion is given, before we end this thesis by presenting some ideas for further development of the LDC support for TCP.

6.1 Summing up

To add LDC support to TCP, a modification to the Linux kernel was made. This modification was designed after a thorough examination of the existing network architecture in Linux. After the implementation was done, several tests were run to evaluate the implementation. The tests show that LDC support in TCP increase the amount of relevant data that can be delivered, by dropping data no longer relevant due to time constraints.

6.2 Conclusion

In this report we wanted to examine if we with LDC support in TCP could reduce latency and increase throughput for time critical data in congested networks. The results presented in chapter 5 clearly states that we can deliver more relevant data with TCP with LDC than with TCP without LDC. However, we can not document any reduced per packet latency, nor document any increased throughput. Actually, we have measured a bit lower throughput with LDC than without. What we have observed is that we by dropping outdated data, are able to send more usable data with LDC. Thus, the *perceived latency* for the receiver will be lower, since we, as an example, always get the whole base layer and thereby get a continuous playback. We can for the same reason claim that we have a better utilization of the throughput, giving us a higher *useful throughput* with LDC than without. Based on this, we conclude that LDC support in TCP actually reduce the latency and increase the throughput for time critical data in congested networks.

6.3 Further work

In this section some ideas for further development of the LDC support for TCP are presented and discussed. One may want to expand the functionality of the LDC support timestamping of packets and give them deadlines. If they are not sent within the deadline, then the kernel can drop the packets automatically, like in SCTP [12]. It may also be desirable to include dropping of retransmissions. However, this implies a modification to the receiving side of a connection due to sequence numbers. In SCTP you have mechanisms to tell the receiver to no longer wait for the given sequence numbers. Similar mechanisms has to be implemented in TCP to support dropping of retransmissions.

An idea for further testing of the LDC support presented in this report, is to implement and run tests where one modifies data in the packet ring directly, instead of just dropping packets. The needed functionality to achieve this is implemented but not tested.

If the LDC support is to go into the Linux kernel production tree, it would be wise to do more testing and maybe do some improvements and optimizations in some places.

Appendix A

Source code

This appendix lists source code for some of the TCP LDC functions mentioned in this thesis. For the full source, please refer to the provided CD-ROM described in appendix B.

A.1 tcp_set_ldc()

```
/* This function is called by tcp_setsockopt() to init
 * the LDC support for the socket.
 */
int _tcp_set_ldc(struct sock *sk, char __user *optval, int optlen)
{
    DPRINT((KERN_EMERG "TCP_LDC: [%lu] ** _tcp_set_ldc(struct sock *
       sk = 0x%lx, char __user *optval = 0x%lx, int optlen = %d)\n",
        jiffies,
        (long unsigned int)sk,
        (long unsigned int)optval,
        optlen));
    int ret;
    int usr_vaules[2];

    copy_from_user(usr_vaules, optval, optlen);

    if (usr_vaules[0] < 0 || usr_vaules[1] < 0)
        return -EINVAL;

    if ((sk->ldc_ring = kmalloc(sizeof(struct ldc_packet_ring),
       GFP_KERNEL)) == NULL)
        return -ENOMEM;

    DPRINT((KERN_EMERG "TCP_LDC: [%lu] Got ring_size: %d and
       packet_size: %d from app\n",
        jiffies,
        usr_vaules[0],
        usr_vaules[1]));
    ret = ldc_pkt_buffer_alloc(sk->ldc_ring, usr_vaules[0],
       usr_vaules[1]);
    if(ret){
        DPRINT((KERN_EMERG "TCP_LDC: [%lu] Got %d from
            ldc_pkt_buffer_alloc()\n",
             jiffies,
             ret));
        return ret;
    }

    sock_set_flag(sk, SOCK_LDC);

    return 0;
}
```

A.2 ldc_pkt_buffer_alloc()

```
/* The follwing function is based on arch/ia64/kernel/perfmon.c and
 * http://www.ussg.iu.edu/hypermail/linux/kernel/0512.0/1038.html
 *
 * This function will allocate a ldc packet buffer and remap it into
 * the user address space of the task
 */
static int ldc_pkt_buffer_alloc(struct ldc_packet_ring *lpr, int
   rsize, int psize)
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *vma = NULL;
    unsigned long size;
    void *pkt_buf;
    int buffer_size, i;


    /*
     * the fixed header + requested size and align to page boundary
     */
    buffer_size = sizeof(struct ldc_ring_head)
        + (rsize * sizeof(struct ldc_packet))
        + (rsize * psize);
    size = PAGE_ALIGN(buffer_size);

    DPRINT(("packet buffer rsize=%d size=%lu bytes\n", buffer_size,
       size));

    /*
     * We do the easy to undo allocations first.
     *
     * ldc_rvmalloc(), clears the buffer, so there is no leak
     */
    pkt_buf = ldc_rvmalloc(size);
    if (pkt_buf == NULL) {
        DDPRINT(("Can't allocate packet buffer\n"));
        return -ENOMEM;
    }

    DPRINT(("pkt_buf @%p\n", pkt_buf));

    /* allocate vma */
    vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!vma) {
        DDPRINT(("Cannot allocate vma\n"));
        goto error_kmem;
    }
    memset(vma, 0, sizeof(*vma));

    /*
     * partially initialize the vma for the packet buffer
     */
    vma->vm_mm          = mm;
    vma->vm_flags       = VM_READ | VM_WRITE | VM_MAYREAD | VM_MAYWRITE
        ;
    vma->vm_page_prot = PAGE_SHARED;

    /*
     * Now we have everything we need and we can initialize
     * and connect all the data structures
     */

    lpr->ldc_pkt_buffer = pkt_buf;
    lpr->ldc_pkt_size = size; /* aligned size */
    lpr->ring_head = pkt_buf;
    lpr->packet_ring = pkt_buf + sizeof(struct ldc_ring_head);
    /* Set the buffer size (in packets) */
    lpr->ring_head->ring_size = rsize;
    lpr->ring_head->packet_size = psize;
    DPRINT((KERN_EMERG "TCP_LDC: [%lu] Init the ldc_ring_head\n",
        jiffies));
    lpr->ring_head->kern_i = 0;
    lpr->ring_head->kern_flag = 1;
    lpr->ring_head->umod_i = 0;
    lpr->ring_head->user_i = 0;

    /*
     * Let's do the difficult operations next.
     *
     * now we atomically find some area in the address space and
     * remap the buffer in it.
```

```
  */
down_write(&current->mm->mmap_sem);

/* find some free area in address space, must have mmap sem held
    */
vma->vm_start = get_unmapped_area(NULL, 0, size, 0, MAP_PRIVATE|
   MAP_ANONYMOUS);  /* Do we need MAP_SHARED and not MAP_PRIVATE
   ? */
if (vma->vm_start == 0UL) {
    DPRINT(("Cannot find unmapped area for size %ld\n", size));
    up_write(&current->mm->mmap_sem);
    goto error;
}
vma->vm_end = vma->vm_start + size;
vma->vm_pgoff = vma->vm_start >> PAGE_SHIFT;

DPRINT(("aligned size=%ld, hdr=%p mapped @0x%lx\n", size, pkt_buf
   , vma->vm_start));

/* can only be applied to current task, need to have the mm
   semaphore held when called */
if (ldc_remap_buffer(vma, (unsigned long)pkt_buf, vma->vm_start,
   size)) {
    DPRINT(("Can't remap buffer\n"));
    up_write(&current->mm->mmap_sem);
    goto error;
}

/* Init the packets */
for(i = 0; i < rsize; i++){
    lpr->packet_ring[i].data = (void *)vma->vm_start + sizeof(
       struct ldc_ring_head) + (rsize * sizeof(struct
       ldc_ring_head)) + (psize * i);
    lpr->packet_ring[i].kdata = pkt_buf + sizeof(struct
       ldc_ring_head) + (rsize * sizeof(struct ldc_ring_head)) +
       (psize * i);
}

/*
 * now insert the vma in the vm list for the process, must be
 * done with mmap lock held
 */

insert_vm_struct(mm, vma);
mm->total_vm  += size >> PAGE_SHIFT;
vm_stat_account(vma->vm_mm, vma->vm_flags, vma->vm_file,
          vma_pages(vma));
up_write(&current->mm->mmap_sem);
```

```
    /*
     * keep track of user level virtual address
     */
    lpr->ldc_pkt_buffer_uaddr = (void *)vma->vm_start;

    return 0;

 error:
    kmem_cache_free(vm_area_cachep, vma);
 error_kmem:
    ldc_rvfree(pkt_buf, size);

    return -ENOMEM;
}
```

A.3 ldc_notify()

```
/* This function get called when the user wants to
 * notify the kernel about new packets in the ring
 */
int _ldc_notify(struct sock *sk)
{
    DPRINT((KERN_EMERG "TCP_LDC: [%lu] ** _ldc_notify(struct sock *sk
        = %lx)\n", jiffies, (long unsigned int)sk));
    /* Clearing the flag */
    sk->ldc_ring->ring_head->kern_flag = 0;

    return ldc_sendmsg(sk);
}
```

A.4 ldc_next_pkt()

```
/* This function finds the next packet to
 * send. Returns -1 if the buffer is empty.
 */
int ldc_next_pkt(struct sock *sk)
{
    DPRINT((KERN_EMERG "TCP_LDC: [%lu] ** _ldc_next_pkt(struct sock *
       sk = %lx)\n", jiffies, (long unsigned int)sk));
    int ret = -1;
    struct ldc_packet_ring *lpr = sk->ldc_ring;
    int pkt = lpr->ring_head->kern_i;

    /* Skiping deleted packets */
    while(LDC_BUFFER_NOT_EMPTY(lpr->ring_head) && lpr->packet_ring[
       pkt].user_flag){
        DPRINT((KERN_EMERG "TCP_LDC:              Deleting packet:
           user_seq = %d, idx = %d\n", lpr->packet_ring[pkt].user_seq
           , pkt));
```

```
            LDC_INCP(lpr->ring_head, kern_i);
            pkt = lpr->ring_head->kern_i;
        }

        if(LDC_BUFFER_NOT_EMPTY(lpr->ring_head)){
            /* Yes, we have packet(s) to send!
             * Inccrementing the pointer so we are
             * ready for the next round.
             */
            ret = pkt;
            DPRINT((KERN_EMERG "TCP_LDC:            Sending packet:
               user_seq = %d, idx = %d\n",lpr->packet_ring[pkt].user_seq,
                pkt));
            LDC_INCP(lpr->ring_head, kern_i);
        } else {
            /* No packet(s) to send.
             * Setting the kern_flag.
             */
            DPRINT((KERN_EMERG "TCP_LDC: [%lu] *  B U F F E R  E M P T Y
               !  *  Setting the kern_flag      *\n", jiffies));
            lpr->ring_head->kern_flag = 1;
        }

        return ret;
}
```

A.5 ldc_update_send_head()

```
/* This function is called by update_send_head() if the current
 * socket is a LDC-socket.
 */
struct sk_buff *_ldc_update_send_head(struct sock *sk, struct
   tcp_sock *tp,
                       struct sk_buff *skb)
{
     DPRINT((KERN_EMERG "TCP_LDC: [%lu] ** _ldc_update_send_head(
        struct sock *sk = %lx, struct tcp_sock *tp = %lx, struct
        sk_buff *skb = %lx)\n",
         jiffies,
         (long unsigned int)sk,
         (long unsigned int)tp,
         (long unsigned int)skb));

      /* The regular update_send_head first we check if we have a
       * pakcet at skb->next. For LDC sockets packets will end up
       * the if the regular write/send etc is used to send packets
       * or if we have a packet with size > mss and the packet was
       * spilt into several skbs. If there was no packet at
       * skb->next and the socket is a TCP LDC socket, this funtion
     * will be called.
     */

    struct sk_buff * new_skb;
    int pkt = ldc_next_pkt(sk);

    if(pkt < 0) {
        /* No packets in the LDC packet ring. */
        new_skb = NULL;
    } else {
        /* We have packets in the LDC packet ring. Make skb and
         * insert it into the send buffer (return to
         * update_send_head).
         */
        new_skb = ldc_make_new_skb(sk, pkt);
        if(!new_skb) {
            /* Something failed. Perhaps memory?
             * Decrementing kern_i and setting kern_flag
             */
            DPRINT((KERN_EMERG "TCP_LDC: [%lu] ldc_make_new_skb()
               FAILED! DECR packet pointer and setting the kernel
               flag.\n", jiffies));
            LDC_DECP(sk->ldc_ring->ring_head, kern_i);
            sk->ldc_ring->ring_head->kern_flag = 1;
        }
    }

    DPRINT((KERN_EMERG "TCP_LDC: [%lu]    ldc_update_send_head()
       returns %lx\n", jiffies, skb));
    return new_skb;
}
```

Appendix B

The CD-ROM

A CD-ROM is attached to the back-cover of this thesis. It contains all the source code for the TCP LDC implementation and test applications. The CD-ROM has the following diretory structure:

```
/
|- src/
   |- linux-2.6.15.4-tcpldc/ - A full Linux source tree with
   |                           the LDC modifications
   |- test_apps/             - The test applications
   |- tcpldclib/             - The user space library
   |- patch/                 - A kernel patch for 2.6.15.4
                               that provides LDC support
```

Enjoy!

Bibliography

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[2] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), April 2004.

[3] Arne Georg Gleditsch and Per Kristian Gjermshus. Cross-referencing linux. `http://lxr.linux.no/`, Jan 2006.

[4] Asvin Goel, Charles Krasic, Kang Li, and Jonathan Walpole. Supporting low latency TCP-based media streams. *Proceedings of IWQoS*, May 2002.

[5] Audio-Video Transport Working Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889 (Proposed Standard), January 1996. Obsoleted by RFC 3550.

[6] S. Hemminger. Network emulation with netem. `http://linux-net.osdl.org/index.php/Netem`, 2005.

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

[8] Charles Krasic, Jonathan Walpole, and Wu-Chi Feng. Quality-adaptive media streaming by priority drop. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 112–121, New York, NY, USA, 2003. ACM Press.

[9] Junwen Lai and Eddie Kohler. Efficiency and late data choice in a user-kernel interface for congestion-controlled datagrams. volume 5680, pages 136–142. SPIE, 2005.

[10] Linux Kernel Programmers. The linux kernel (2.6.15.4). `http://www.kernel.org/`, Feb 2006.

[11] Miguel Rio, Tom Kelly, Mathieu Goutelle, Richard Hughes-Jones, and Jean-Philippe Martin-Flatin. A Map of the Networking Code in Linux Kernel 2.4.20. Technical Report DataTAG-2004-1, DataTag project (IST-2001-32459), March 2004.

[12] 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.

[13] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. Updated by RFC 3309.

[14] Klaus Wehrle, Frank Pahlke, Hartmut Ritter, Daniel Muller, and Marc Bechler. *Linux Network Architecture*. Prentice Hall, 2004.