BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

IMPLEMENTATION OF DELTA-T TRANSPORT PROTOCOL

IMPLEMENTACE TRANSPORTNÍHO PROTOKOLU DELTA-T

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. ZDENĚK CHOVANEC**
AUTOR PRÁCE

SUPERVISOR **Ing. MICHAL KOUTENSKÝ**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2021/2022

Master's Thesis Specification

Student: **Chovanec Zdeněk, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Cybersecurity
Title: **Implementation of Delta-t Transport Protocol**
Category: Networking
Assignment:

1. Carefuly study the design of Delta-t protocol.
2. Familiarize yourself with the architecture of the Linux kernel, its implementation of the networking stack and usage of kernel modules.
3. Design a Delta-t module for the Linux kernel.
4. Implement the design from point 3.
5. Evaluate the functionality of reliable network transmission when using the implemented protocol.
6. Measure, analyze, and compare the implemented protocol's properties with TCP.

Recommended literature:

- Watson, R.. "Timer-Based Mechanisms in Reliable Transport Protocol Connection Management." *Comput. Networks* 5 (1981): 47-56.
- Love, Robert. *Linux kernel development*. Upper Saddle River, NJ: Addison-Wesley, 2010.
- Rosen, Rami. *Linux Kernel Networking: Implementation and Theory*. New York, N.Y: Apress, 2014.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Koutenský Michal, Ing.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: October 25, 2021

Abstract

Delta-t protocol represents a network transport protocol that is currently available only as a specification. The aim of this work is to implement the protocol as a Linux kernel module, extending the current TCP/IP stack. Delta-t supports reliable, connection-oriented, full duplex communication between two endpoints. Main contribution of the protocol is in the area of connection management. No extra packet exchanges are needed for that purpose. Delta-t employs timer-based mechanism in order to avoid connection hazards. Apart from connection management, Delta-t and TCP are quite similar. Therefore, comparison of the two will be provided.

Abstrakt

Transportní protokol Delta-t se v současné době vyskytuje pouze ve formě návrhu. Cílem této práce je vytvoření implementace protokolu v prostředí Linuxového TCP/IP zásobníku. Implementace bude mít formu zásuvného modulu. Protokol Delta-t patří mezi protokoly zajišťující obousměrný spolehlivý přenos. Protokol přichází s velmi jednoduchým a efektivním způsobem správy spojení. Spolehlivé sestavení a ukončení spojení není realizováno prostřednictvím *handshake* zpráv. Aktuální stav spojení je dán pouze dobou, která uběhla od posledního přijetí či odeslání datového segmentu. V ostatních aspektech se Delta-t podobá protokolu TCP, a proto budou jejich implementace vzájemně porovnány.

Keywords

computer networks, Delta-t protocol, transport layer, connection-oriented protocol, connection management, timer-based mechanism, reliable communication, TCP/IP stack, Linux, loadable kernel module

Klíčová slova

počítačové sítě, protokol Delta-t, transportní vrstva, spojovaně orientovaný protokol, správa spojení, časovače spojení, spolehlivý přenos, TCP/IP zásobník, Linux, kernelový modul

Reference

CHOVANEC, Zdeněk. *Implementation of Delta-t Transport Protocol.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Koutenský

Rozšířený abstrakt

Ještě v nedávné minulosti byly informace šířeny pouze skrze televizní a rádiové vysílání, noviny, či pomocí mluveného slova. To se však v posledních dekádách razantně změnilo. Inovace v oblasti elektroniky, zejména v polovodičové technologii, zapříčinily postupné pronikání počítačových systémů do našich životů. Tyto počítačové systémy jsou reprezentovány například chytrými telefony, televizemi, laptopy či herními konzolemi. Všechna tato zařízení mají jedno společné, a to schopnost vzájemně komunikovat. Tuto schopnost jim zprostředkovávají počítačové sítě. Nejproslulejší počítačovou sítí je Internet, jenž se skládá z obrovského počtu vzájemně propojených zařízení.

V každé síti, včetně Internetu, se můžeme setkat s celou řadou protokolů. Protokoly lze dle jejich vlastností roztřídit do několika skupin, které označujeme jako vrstvy. Protokol formálně definuje, jakým způsobem má vypadat určitá zpráva v podobě posloupnosti bitů a bajtů. Mimoto může protokol také udávat, jakým způsobem by měl systém reagovat při přijetí nebo odeslání dané zprávy.

Internet tvoří rodina protokolů označovaná jako TCP/IP sada nebo zkráceně TCP/IP. Tato zkratka je spojením názvů dvou nejvýznamnějších protokolů, a to protokolu TCP (*Transmission Control Protocol*) a protokolu IP (*Internet Protocol*). Protokol IP zajišťuje doručování síťových paketů z jednoho konce sítě na druhý, z jednoho koncového zařízení do druhého. Pakety se však po cestě mohou ztratit, poškodit anebo dorazit k příjemci vícekrát (duplikace) či v jiném pořadí, než byly odeslány. Protokol IP nedokáže zaručit spolehlivé doručení.

Na druhou stranu protokol TCP spolehlivé doručení zaručit dokáže. Zajišťuje totiž spolehlivý přenos dat skrze nespolehlivou síť. Před tím, než mohou být pomocí TCP odeslána data, je nejdříve nutné s protistranou navázat spojení. Proces navazování spojení TCP protokolu se označuje jako třícestný handshake (three-way handshake). Při tomto procesu je nutné odeslat tři segmenty. Spojení je taktéž nutné explicitně ukončit. To může vyžadovat odeslání až čtyř segmentů na samém konci datové komunikace. V jaké fázi se dané spojení právě nachází je interně uloženo ve stavové proměnné. Tato proměnná, s ohledem na procedury navazování a ukončování spojení, může nabývat mnoha hodnot, což má za následek značnou složitost protokolu.

V této práci se budeme věnovat protokolu Delta-t, jenž se v mnoha ohledech podobá protokolu TCP. Zásadně se však liší v přístupu k navazování a ukončování spojení. Navázání a ukončení spojení nevyžaduje odeslání jediného segmentu. Stav aktuálního spojení závisí pouze na době, která uběhla od posledního odeslání či přijetí datového segmentu. Protokol Delta-t v této době existuje pouze na papíře a nemá reálnou implementaci. Našim cílem je implementovat tento protokol jako zásuvný modul Linuxového jádra. Za tímto účelem se nejdříve hlouběji seznámíme s návrhem protokolu a také s Linuxovým síťovým zásobníkem. Rovněž ukážeme klíčové body implementace modulu, kterým rozšíříme množinu podporovaných protokolů o protokol Delta-t. Na závěr Delta-t porovnáme s TCP a dosažené výsledky zhodnotíme.

Implementaci protokolu Delta-jsme porovnávali s TCP v podmínkách, ve kterých jsme simulovali ztrátu a zpoždění paketů. Delta-t bylo v jednom scénáři schopno držet krok s TCP, nicméně v ostatních dvou TCP dominovalo. Při simulaci vysokého procenta paketové ztráty Delta-t selhalo, zatímco TCP zajistilo spolehlivý přenos. Selhání Delta-t bylo zapříčiněno absencí mechanismu řízení zahlcení. Do budoucna by bylo určitě vhodné tento mechanismus začlenit do Delta-t. Obecně vzato modul prezentovaný v této práci toho má ještě mnoho co zlepšit. Cílem této práce však nebylo vytvořit dokonalou implementaci spolehlivého transportního protokolu, nýbrž ověřit koncept protokolu Delta-t v praxi.

Implementation of Delta-t Transport Protocol

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Michal Koutenský. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Zdeněk Chovanec
May 23, 2022

Acknowledgements

I would like to express many thanks to my supervisor Mr. Ing. Michal Koutenský for his guidance throughout the process of making this work, for the extra work he had to do, and for having quite a bit of patince with me. Also, I owe big thanks to my family for their support.

Contents

Chapter 1

Introduction

It was not so long ago, when any new information could be spread only by means of radio/television broadcast, local newspaper, or a spoken word. However, in the past few decades things changed quite noticeably. Unceasing developments in the area of electronics, especially in semiconductor technology, led to slow but steady proliferation of computer systems into our lives. These computer systems are represented by a wide variety of devices such as smartphones, TVs, laptops, gaming consoles and many others. There is one thing that all these devices have in common. It's the ability to communicate with each other and that is the place where computer networks come into play.

The most well-known example of a computer network is the Internet. It consists of an astrological number of interconnected devices. The Internet plays a substantial role in our day-to-day routines. Accessing video streaming platforms, World Wide Web, using instant messaging apps and electronic mail, represent just the tip of the iceberg of the possible uses of the Internet. But how does that Internet thing work?

When describing computer networks, two closely related terms will almost certainly pop up – protocols and layers. For computer systems to communicate, there needs to be an agreement ahead of time on the order and format of individual messages. Network protocols are used for this purpose. They formally describe exact sequences of bits and bytes that constitute given message. They can also mandate what actions should be taken upon reception or transmission of such a message.

A layer is an abstract concept that enables us to group protocols based on their properties. Individual layers are stacked together. Starting from the bottom, each subsequent layer (and corresponding protocol) takes care of different aspect of network communication. Upper layer protocols rely on the services provided by the layers below it and usually add some functionality on top. As a result, an action as simple as sending an email message requires the message to pass through multiple layers, which in turn involves a cooperation of multiple network protocols.

Although the range of protocols that constitute today's Internet is quite broad, there are two that stand out. Its Transmission Control Protocol (TCP), specified in RFC 793, and Internet Protocol (IP), specified in RFC 791. IP represents network layer, commonly referred to as layer 3 (L3), protocol. It is responsible for delivering network packets from one host to another, from one side of the network to the other. During this process packets could get lost, damaged or could be delivered in different order than they were originally sent. In other words, IP performs unreliable network packet delivery as it cannot guarantee that packet reached its destination.

TCP on the other hand can give such a guarantee. TCP performs reliable data delivery over unreliable network. TCP belongs to the family of transport layer protocols. Transport layer is in literature often denoted as layer 4 (L4). When using TCP, a connection must be established between client and server prior to any data being sent. In standard mode of operation, it is accomplished by three-way handshake (3WHS). This requires three segments to be sent at the beginning of each connection. In addition to that, connection termination procedure may require up to four segments to be exchanged. In consequence, TCP internally requires a state machine with many states to indicate where in the opening or closing sequence we are currently at. This fact increases the overall protocol complexity.

In this work we will survey Delta-t transport layer protocol. It guarantees reliable delivery, in similar fashion as TCP, but it employs completely different approach when it comes to connection establishment and termination. No segments are exchanged when the data transfer is about to start or after it finishes. In a nutshell, the current state of a connection is based exclusively on the amount of time that has passed since last data segment was sent or received.

For any network protocol to be of any use it must have a software implementation. The implementation of standard Internet protocols is called TCP/IP stack. It is usually included in the kernel of the operating system of our choice. In contrast with TCP, no implementation of Delta-t has been preserved to this day.

The primary goal of this work is to make a proof-of-concept implementation of Delta-t transport protocol. The freely available Linux kernel platform was chosen as implementation target. Delta-t support will be added in the form of loadable kernel module.

First, we need to familiarize ourselves with the ideas behind the Delta-t protocol. We can do so in chapter 2. In chapter 3 a brief introduction of Linux TCP/IP stack and its core data structures can be found. Chapter 4 will first lay out the plan four our work – the design of Delta-t kernel module will be presented. After that, the most important components of Delta-t module implementation are mentioned. Chapter 5 evaluates the implemented protocol in different scenarios and compares it with TCP. Finally, chapter 6 contains the conclusion of my work.

Chapter 2

Delta-t Protocol

This chapter describes Delta-t protocol and its major features. Primary source of information is the working draft of Delta-t protocol specification [18]. Before we begin, it is necessary to point out that Delta-t, unlike TCP, was not designed to operate on top of IP. Instead, it uses services provided by another connectionless network protocol (called *DeltaGram*). In the text to follow only the most relevant facts regarding the protocol are mentioned. Consequently, details such as data units originally used or addressing conventions are omitted from our discussions.

TCP and Delta-t have many similarities. Both are connection-oriented transport layer protocols that provide reliable data transfer. Also, both are flow controlled and allow full-duplex style of communication. The reliability of Delta-t is accomplished by using sequence numbers, positive acknowledgements (acks), retransmission mechanism and, most importantly, bounds on packet lifetime [18].

The primary contribution of Delta-t transport protocol is in the area of connection management [17]. As opposed to other transport protocols, there is no packet exchange taking place to manage a connection. The connection management scheme of Delta-t is based on the use of two timers for each connection.

2.1 Connection Management

When two entities are reliably exchanging data over a network, at each end there is a data structure that keeps state information describing the progress of data transfer. In compliance with [18], we will call this data structure *connection record* (CR), although these days it is commonly referred to as *socket*.

To ensure reliable data delivery, Delta-t assigns sequence number (SN) and checksum to each data segment transmitted. Connection management must properly initialize SNs and update CRs, so that they correctly reflect the state of a connection throughout its lifespan.

There are 3 conceptual phases connection management can be divided into. First, the connection needs to be opened/established. To do that, we need to allocate and initialize a CR. After the initialization, we can start with the actual communication, that is, start exchanging data. During this phase, the CR is frequently updated. Finally, when we have no more data to send we can close the connection and release resources (i.e., free the CR).

To open a connection, we need to choose an initial SN. To perform reliable open the SN cannot be chosen arbitrarily. Facing a network where segments can potentially get

damaged, lost, duplicated, or delivered out of order, we need to choose initial SN so that the following conditions hold (from [19]):

O1: *If no connection exists and the receiver is willing to receive, then no segments from a previously closed connection should cause a connection to be initialized and duplicate data to be accepted.*

O2: *If a connection exists, then no packets from a previously closed connection should be acceptable within the current connection.*

Terminating a connection is another important aspect of connection management. Ideally, each connection should be closed gracefully. This way the connection is not closed until all segments needing acks can be acknowledged. When the connection is closed, each end knows whether the other end received all data sent or not. As a result, the ambiguity about the state of data sent is eliminated when protocol performs graceful close. The following two conditions imply a graceful close of a connection (from [17]):

C1: *A receiving side must not close until it has received all of a sender's possible retransmissions and can respond to them.*

C2: *A sending side should not close until it has received acknowledgement of all that it has sent. In particular it should allow time for an acknowledgement of its final retransmission, if needed, before reporting a failure to its client program.*

Delta-t connection management is purely timer-based. Both ends of a connection need to manage one timer for each direction of data flow. In other words, two timers are needed if duplex communication takes place. Receive-timer (Rtimer) is tied with inbound traffic, while the outbound traffic is under control of Send-timer (Stimer).

The conditions to perform reliable open (conditions O1 and O2) and graceful close (conditions C1 and C2) are satisfied, if the timers are initialized properly and if we follow a set of rules. Safe values to initialize the timers are:

$$\text{Rtimer} = 2 \cdot \Delta t$$

$$\text{Stimer} = 3 \cdot \Delta t$$

where $\Delta t = \text{MPL} + \text{R} + \text{A}$

- MPL (Maximum Packet Lifetime) is a worst-case estimate of the time for traversing the network.
- R is the maximum time a sender will keep retransmitting a segment.
- A is the maximum time a receiver will wait before sending (delayed) ack.

Additionally, this set of rules must be applied:

R1: *Stimer is refreshed whenever a new SN (i.e., a new data or rendezvous segment) or reliable-ack is sent.*

R2: *Once a segment s_i has had its maximum retransmission time (or equivalently maximum number of retransmissions) no new segments can be transmitted until s_i has been acked; segments s_{i+k} which had previously been transmitted can continue being retransmitted until their maximum retransmission time.*

R3: *Rtimer refreshed whenever a new SN is accepted, or data overflow occurs.*

R4: *When Rtimer expires, the receive state is reset to its default values.*

R5: *Once a data or rendezvous or reliable-ack segment is initially transmitted its lifetime is set equal to* `Δt` *and starts counting down.*

R6: *At the point an SN is tested for acceptance, the lifetime of any ack segment generated is set equal to* `Δt` *and begins counting down.*

R7: *When Stimer expires the send state is reset to its default values, any initial SN can be used when new data needs sending, and if unacked SNs exist an error is reported.*

There are no dedicated segments for connection opening or closing in Delta-t. Sending the first segment causes the sender's Stimer to be initialized and the connection to be opened. The reception of this segment opens the receiver's connection and leads to Rtimer initialization. The connection is closed (or put equivalently — restored to default state) when both Stimer and Rtimer reach zero. If Stimer goes to zero, then any initial SN for an outgoing segment can be used. If it is nonzero, then the next SN expected by the receiver must be applied. Receiver must only accept segments with SNs in its acceptance window. If the Rtimer is zero, any SN is acceptable.

There is one flaw in the procedure presented so far. The receiver cannot detect out of order segments before initializing its state. To cope with that, Delta-t employs so called *data-run* flag in segment header. This flag is set only if all previously sent segments have been acknowledged. Consequently, when Rtimer is zero only segments with *data-run* flag set are acceptable. An example of unidirectional communication (for the sake of simplicity) using Delta-t protocol is illustrated in figure 2.1.

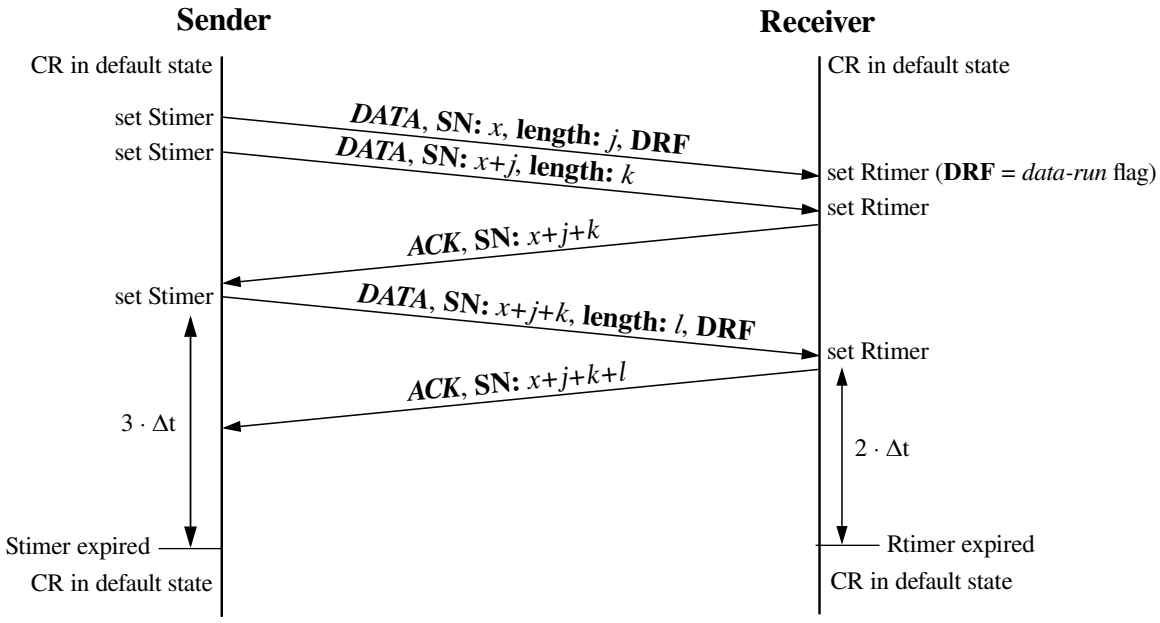

Figure 2.1: Example of Delta-t communication (taken from [18], modified).

Unfortunately, computer systems are subjects of occasional crashes in practice. In this case a graceful close is not an option as any outstanding connection will be shutdown immediately, regardless of connection management mechanism being used.

Proper recovery routines need to be in place to eliminate connection management hazards resulting from a crash/failure of either side of a connection. Due to its timer-based nature the recovery routine of Delta-t has the form of waiting an interval before initiating or accepting a connection.

- After crash, the sender must wait $3 \cdot \Delta t$. After this interval it is guaranteed that any data or ack segments having the same SN have died out and that any initial SN will be accepted by receiver (because Rtimer will not be set by that time).
- Receiver's recovery interval is Δt. It ensures that all segments from previous connection have expired (assuming sender sticks to its recovery interval).

In summary, at most two timers per host are needed for Delta-t's connection management. During unidirectional communication sender's Stimer and receiver's Rtimer are set to $3 \cdot \Delta t$ and $2 \cdot \Delta t$, respectively. Furthermore, it is important to note that in duplex mode the value of Δt might be different for each direction of data movement. There is no packet exchange overhead related to opening or closing a connection. Reliable delivery of any data at any point in time makes use of at most two segments. One segment for the data and one for its ack. If delayed acks are used even better protocol efficiency can be achieved.

2.2 Flow Control

Flow control is a mechanism that systematically deals with senders that want to transmit data faster than the receivers can accept them. This situation can occur if sender is running on fast, powerful computer (e.g., Web server) and the receiver is running on a much slower machine (e.g., smart phone). Soon the slower machine would not be able to buffer incoming segments and start dropping them. [16]

Fortunately, this issue has been thought of when designing the protocol. Delta-t uses sliding window approach to control the flow of data exchange. An implementation of sliding window flow control maintains a set of SNs a sender is permitted to send at any given instant of time [16]. The concrete details of window management, described in Delta-t specification, are skipped because they are no longer relevant from the perspective of this work.

However, there is one important procedure outlined. It is called *rendezvous-at-sender*. It reliably takes care of the situation when sender's sending window shrinks to zero, but there is still some data to send. This situation is solved in a lightweight manner without the need of polling, used for example, in TCP. The procedure works as follows:

When all data sent have been acked and the send buffer is not empty, but sender faces zero window, it sends rendezvous segment. When the segment reaches the other end, it is an indication that the sender wants to be informed when the window opens. The rendezvous segment is transferred reliably meaning that it is assigned a SN and it is retransmitted until acked or until the retransmission interval expires. When the receiver's input window opens, after previously accepting a rendezvous segment, it will send an ack segment with dedicated flag set (window-opening ack). The receiver will retransmit this ack segment until an acceptable data segment is received. The data segment effectively acks the window-opening ack. The window-opening ack is in Delta-t called reliable-ack because it has the *reliable* flag set in segment header.

Right now, the *rendezvous-at-sender* procedure is straightforward: send rendezvous segment when all previous data segments have been acked and then wait for reliable-ack. The description presented so far is not complete, nonetheless. An issue might occur if the

amount of data sent could not fit in the receiver's advertised window. This situation is called window overrun. Data that overflown the window will never be acked. Subsequently, the rendezvous segment would never be sent.

Luckily, Delta-t offers a solution to this issue. An event of window overrun is signaled to sender via *overflow* flag in ack segment header. Once an ack with the *overflow* flag set is received the state of segments that exceeded the window is reset as if they were never sent. As a result, no segments are unacked, and the rendezvous segment can be generated.

Nevertheless, a potential hazard exists still. Duplicates of overrun segments might ack a reliable-ack. If the reliable-ack and the ack of duplicate data (i.e., overrun segment) just accepted by the receiver are both lost, then the sender will never get to know that the window has opened.

The countermeasure to this problem employed in Delta-t is to use a SN offset. The sender chooses sufficient offset so that when it is added to the current SN it yields a SN larger than any SN already sent. The offset value is put in rendezvous segment. SN assigned to the rendezvous segment is the same as the one in the last ack segment so that the receiver accepts the rendezvous segment. After the rendezvous segment has been transmitted, sender shifts its SNs by the offset. Once the rendezvous segment is received, receiver reads the offset value and shifts its input window SNs as well. *Rendezvous-at-sender* procedure, described in previous paragraphs, is illustrated in figure 2.2.

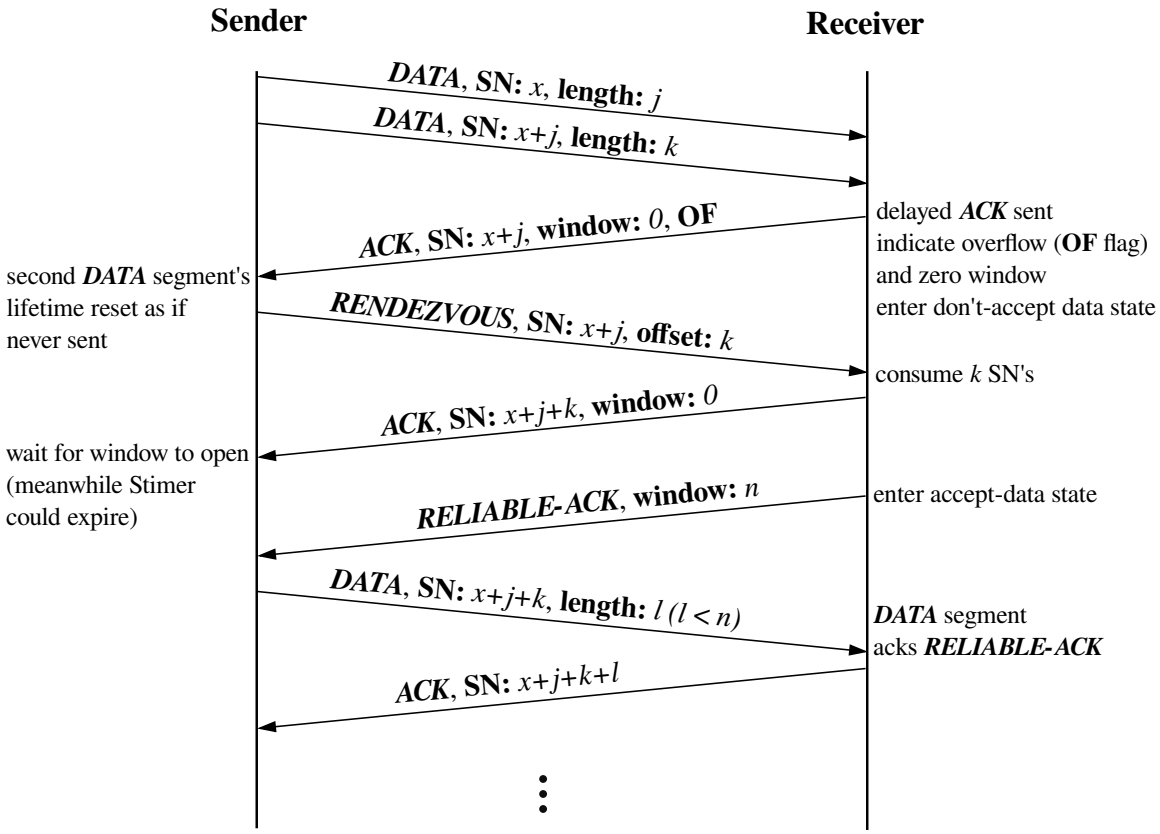

Figure 2.2: Rendezvous-at-sender procedure with window overrun (from [18], modified).

2.3 Congestion Control

Each computer network is limited in terms of the amount of data it can handle. There are some unpleasant consequences of putting a network under heavy load. To name a few, decrease of the overall throughput, increase of packet delay and even a packet loss can be observed. Exceeding the network capacity inevitably leads to congestion. Congestion control (CC) mechanisms are trying to solve the problem of using the network as efficiently as possible while achieving the highest possible throughput with minimal loss ratio and small delay [20].

Delta-t protocol specification lacks any CC mechanism. We must bear in mind that at the time the protocol was designed network congestion was not considered as troublesome area that could potentially cause serious issues. RFC 793 (TCP) from 1981 did not mention any CC algorithm either. However, the widespread use of TCP without any CC algorithm did not last very long.

In October 1986 congestion collapse happened. It had been observed that important component of the early Internet operated 1000 times worse than it should have due to aggressive retransmission caused by packet loss. The network became persistently congested resulting in massive packet loss and low throughput. [4]

Nowadays, the need for congestion control/avoidance algorithms in transport layer protocols is clearly more understood. They are incorporated in TCP, DCCP and SCTP protocols and it would be desirable to utilize them also in Delta-t.

2.4 Services of Lower Layers

In section 2.1 we introduced the theory behind reliable connection management using timers. However, for this theory to work it is crucial that we have some way to bound segment lifetime. Moreover, we need to bound this lifetime in units of time rather than routing hops.

In Delta-t, lifetime field in network layer header is used to limit packet lifetime. Delta-t usually initializes the value of this field to `Δt` when new segment is being formed. The underlying network then performs packet aging services and discards packets whose lifetime has expired. This must be accounted for during the design of Delta-t Linux kernel module.

Chapter 3

Linux Kernel

Linux kernel is the at the core of any Linux operating system. The most common and probably the easiest way to run Linux kernel is by installing a Linux distribution (e.g., Debian, Ubuntu, Fedora and so on). These distributions package the Linux kernel with other utilities to suit the needs of most users. In the rest of this work the term Linux will be referring to Linux kernel.

Linux was created by a Finnish student, Linus Torvalds, back in 1991. Over the years it has proven to be reliable, stable and its popularity has started to grow. These days we can encounter Linux operating system in data centers, embedded devices like wireless routers, set-top boxes, or medical instruments. Moreover, the Android operating system, which can be found in many smartphones and smart TVs, is based on the Linux kernel. [12]

One of the key characteristics of Linux is that it's not a proprietary product. It is open source software licensed under GNU General Public License (GPL). As a result, anyone is free to download the source code and make changes to it. [6]

The fact that Linux is easily modifiable, free, and open source makes it perfect fit for education and academic purposes. However, there is one downside to it. Linux code base is very large (millions of lines of code) [2]. In addition to that, due to the continual improvements and performance optimizations the learning curve for newcomers might be very steep.

Fortunately, as we are concerned with implementing new transport layer protocol, we can leave the code dealing with process scheduling, memory management and other core internals aside and focus only on the networking subsystem.

3.1 Socket API

The networking subsystem is necessary for any sort of network communication, but by itself is not sufficient. What is required in addition, is an interface between the networking subsystem and user space applications. Without this interface the kernel and user space programs could not interoperate. Linux implements the POSIX socket API, which was specified by the IEEE in POSIX.1g [12]. This API is also called BSD or *Berkeley* sockets API because it originated with the 4.2BSD system, introduced in 1983 [14]. Through the use of commonly known socket API functions (`socket()`, `bind()`, `listen()`, `connect()`, `accept()`, etc.) we can instruct the kernel to issue network related operations on our behalf.

When opening a socket, we can choose from multiple socket types. Socket type defines the semantics of the communication. The assignment of a socket type to transport layer

protocol is based on protocol properties. From the perspective of this work, the following socket types are of interest:

- stream socket (type `SOCK_STREAM`) provides reliable, sequenced, full-duplex octet streams between the socket and a peer to which the socket is connected. ... Record boundaries are not maintained; data sent on a stream socket using output operations of one size may be received using input operations of smaller or larger sizes without loss of data. Data may be buffered; successful return from an output function does not imply that the data have been delivered to the peer or even transmitted from the local system. If data cannot be successfully transmitted within a given time, then the connection is considered broken, and subsequent operations shall fail. ... [5]

- sequenced stream socket (type `SOCK_SEQPACKET`) is similar to the `SOCK_STREAM` type, and is also connection-oriented. The only difference between these types is that record boundaries are maintained using the `SOCK_SEQPACKET` type. ... Record boundaries are visible to the receiver via the `MSG_EOR` flag in the received message flags returned by the `recvmsg()` function. ... [5]

3.2 Networking subsystem

At the bottom layer of networking subsystem resides network device (represented by `struct net_device` in kernel code). It communicates with other devices using link layer protocol, for example, Ethernet. The network device is managed by device driver. Device driver forwards input packets to the network layer. Output packets, received from the network layer, are checked by device driver and then passed to the network device to send them over the medium.

Network layer is right above the link layer. In the view of this work, it is represented by the IP protocol. Egress packets might get fragmented before being passed to the link layer. In a similar way, IP might reassemble multiple ingress packets into one before being delivered to the transport protocol.

Transport layer processing is the closest one to the user application that is still carried out by the kernel. Depending on the type of traffic (i.e., outbound, or inbound) transport protocol either hands the data over to the application through the socket interface or passes the data directly to the network layer.

Linux networking subsystem is generic enough so that we can omit link layer and network layer specific code from our discussions. However, the transport layer will not be described in detail here. The reason for that is that transport layer processing is implemented by a collection of callback pointers (callbacks). All these callbacks are protocol (e.g., TCP, UDP, etc.) specific. Instead, we will gloss over three key data structures of transport layer code and their most prominent fields. Once we become acquainted with those data structures we can proceed to a brief explanation of transport protocol registration procedure.

3.2.1 Data structures

`struct socket`

BSD sockets are internally represented by `struct socket`. Socket-related system calls work with this structure. This structure has only 7 fields, the most notable are:

- `struct sock *sk` — pointer to the network layer representation of this BSD socket. In the rest of this work the term BSD socket will be used for user-space socket or `struct socket`, while the term NET socket will refer to `struct sock`. NET sockets are further described in the text that follows.
- `const struct proto_ops *ops` — `struct proto_ops` contains a number of callbacks, that implement functions like `connect()`, `listen()`, `sendmsg()`, `recvmsg()`, etc. Implementation of these functions is network family (e.g., `AF_INET`) specific. Transport layer protocols do not share `struct proto_ops`, but what they might share are the callbacks inside. For example, when `sendmsg()` is called on a BSD socket the call will eventually be resolved to `ops->sendmsg`. `ops->sendmsg` of both TCP and UDP points to the `inet_sendmsg()` function. Although these protocols are completely different, the same function will be invoked when new data is being sent. How is this possible will be revealed when we describe NET socket.

`struct sock`

It is the third representation of a socket when going from top to bottom. At the top there is user-level socket, then follows its in-kernel version – `struct socket`. Ultimately, at bottom resides NET socket – `struct sock`, which is the network layer representation of the previous two. It is the place where all the networking-related information is stored. To give an idea, here are some of its members:

- `struct socket *sk_socket` — the BSD socket associated with this NET socket.
- `int sk_sndbuf` — the size of the send buffer (write queue) in bytes.
- `int sk_rcvbuf` — the size of the receive buffer (receive queue) in bytes.
- `struct sk_buff_head sk_receive_queue` — a queue of incoming packets.
- `struct sk_buff_head sk_write_queue` — a queue for outgoing packets.
- `atomic_t sk_drops` — number of packets dropped by this socket.
- `struct proto *sk_prot` — this structure contains mostly callbacks. These callbacks implement transport layer protocol-specific actions. Continuing with our discussion about the `sendmsg()` socket function call, we will show the implementation of `inet_sendmsg()`. Definition of the function is presented in figure 3.1[1].

```
int inet_sendmsg(struct socket *sock, struct msghdr *msg,
                 size_t size)
{
        struct sock *sk = sock->sk;
        ...
        return sk->sk_prot->sendmsg(sk, msg, size);
}
```

Figure 3.1: Definition of the `inet_sendmsg()` function.

[1]Note that the code snippet was simplified, and a portion of the code was left out (...).

First, the NET socket is acquired by reading the `sk` field of a BSD socket. Next, the protocol-specific callback is invoked. For TCP `sk->sk_prot->sendmsg` points to `tcp_sendmsg` and for UDP to `udp_sendmsg`. In conclusion, every elementary socket function will eventually call the right protocol handler by means of a chain of callbacks.

`struct sk_buff`

This structure is ubiquitous to Linux networking subsystem. The socket buffer alias `struct sk_buff` (SKB) represents any incoming or outgoing packet, including the headers, the payload and other related information [12]. The most noteworthy members are the following:

- `struct sock *sk` — pointer to the NET socket that is associated with this packet.
- `unsigned char *head`
- `unsigned char *data`
- `sk_buff_data_t tail`
- `sk_buff_data_t end`
  - These four fields are closely interrelated. `head` points to the start of the linear data area of a SKB. In this area packet data and headers can be stored. `end` marks the end of this area (`sk_buff_data_t` type optimizes memory usage – an offset is kept instead of a pointer). Our current position in the buffer is held in `data` pointer. End of the actual packet data is delimited by `tail`. To aid with the description, an illustration of a SKB a) right after it has been allocated and b) right before it is handed over to the device driver is provided in figure 3.2.

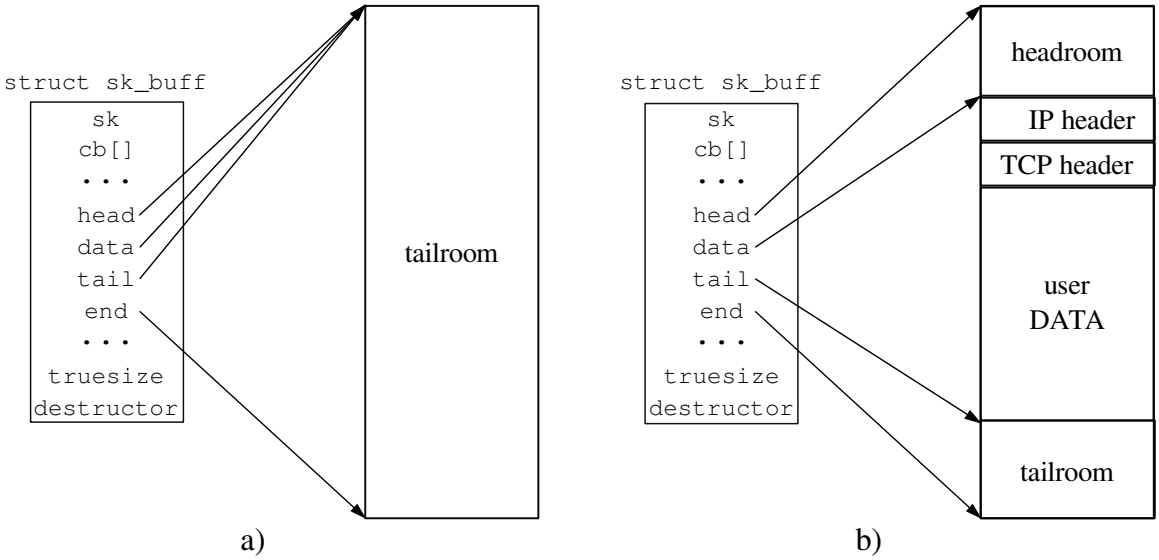

Figure 3.2: `struct sk_buff` a) freshly allocated b) after leaving the network layer.

- `char cb[48]` — this is where per packet control information is stored. Transport layer protocols are free to use this space as desired. For example, TCP stores here segment SN as well as acknowledgement SN in host byte order. Using this approach, the SN stored in TCP header does not have to be accessed and converted every time some function needs it.
- `unsigned int truesize` — total size of a SKB. It sums the size of the data area we allocated for the packet and the size of the `struct sk_buff` itself.
- `void (*destructor)(struct sk_buff *skb)` — function callback to invoke when we free this socket buffer.

3.2.2 Transport protocol registration

When a network packet is received, an array of 256 records is indexed by the value of *protocol* (IPv4) or *next header* (IPv6) packet header field. The selected record contains handler for the incoming packet. New protocol is added to the array using `inet_add_protocol()`. When an invocation of this function succeeds, the kernel is ready to accept and process packets carrying the new protocol.

There is no way to generate traffic, however. No sockets matching the new protocol can be created. A remedy for this situation is the `inet_register_protosw()` function. It takes `struct inet_protosw` as its only argument. The protocol registration will be discussed more thoroughly in implementation section where we will include snippets of actual code.

In conclusion, only two function calls are required to register new transport layer protocol. Nevertheless, a great amount of work must be done to properly initialize the arguments of those two function calls.

3.3 Loadable kernel modules

Linux is a monolithic kernel. The entire kernel code runs in special, privileged mode called *kernel mode*. Ordinary applications run in *user mode*. When code running in *user mode* (e.g., some user application), starts to misbehave, the kernel has the ability to shut it down. Nonetheless, kernel is not under any sort of supervision. Faulty operation performed by the kernel might seriously damage or even crash the entire system. Key feature of Linux is its ability to dynamically load separate binaries, called loadable kernel modules (LKMs), into the kernel image. [6]

These modules enable us to extend the functionality of the kernel without the need to rebuild and reboot it every time we make any changes. Thus, the development process can be faster [13]. However, there is one downside to it. Monolithic kernels typically exist on disk as a single static binary [6]. In other words, all source files of Linux kernel are linked into a single base image. As a result, built-in kernel code can call any nonstatic function (assuming it is declared in one of the included header files). For LKMs the situation is different. Only explicitly exported functions are available. Specifically, kernel symbol must be enclosed in `EXPORT_SYMBOL` or `EXPORT_SYMBOL_GPL` macros to be usable from external module.

Example of a trivial LKM is demonstrated in figure 3.3. Every module must implement initialization and cleanup function. In older kernels, it was necessary to name these functions `init_module()` and `cleanup_module()`, respectively. Nowadays, the preferred

method is to use a name of our choosing and then use `module_init()` and `module_exit()` macros. For the module to compile, license must be specified.

```c
#include <linux/module.h> /* Needed by all modules */

static int my_init_fn(void)
{
    pr_info("Init done!\n");
    return 0;
}

static void my_cleanup_fn(void)
{
    pr_info("Cleanup done!\n");
}

module_init(my_init_fn);
module_exit(my_cleanup_fn);

MODULE_LICENSE("GPL");
```

Figure 3.3: Example of a (trivial) loadable kernel module.

Kernel modules, as well as the whole Linux kernel, are built by the *kbuild* build system. Usually, a simple *makefile* with few special directives will suffice to build the module. However, the prerequisite for the build is that we have the kernel headers available.

Chapter 4

Delta-t kernel module

Having introduced the Delta-t protocol and Linux networking infrastructure in chapters 2 and 3, we can proceed to the next step. It is time to translate the concepts surrounding Delta-t, currently expressed only on paper, into a loadable kernel module. This module will extend the set of supported transport layer protocols. IP packets will carry Delta-t segments.

Unfortunately, Delta-t specification in its current format is not compatible with IP. More specifically, the connection management hazards might not be prevented. In the following sections a solution to this problem will be presented and a foundation for our work will be laid out.

4.1 Design

In IP networks the lifetime of a packet is kept in *time-to-live/hop count* field of IPv4/IPv6 header. Value stored in this field limits the number of routing hops that a given packet may traverse. Delta-t segment lifetime must be bound in units of time, rather than by the number of routing hops. Only this way the connection management hazards will be avoided (assuming the rules and recovery intervals from chapter 2 are followed and timers are initialized accordingly). New network layer protocol, capable of packet aging services, similar to the one used originally alongside Delta-t, would solve this issue. In this work, a different solution will be outlined.

4.1.1 Segment Lifetime

Introducing lifetime and timestamp field into Delta-t segment header could potentially resolve our problem. The sender would set the lifetime and fill in the timestamp field with current time before sending a segment. The receiver would be able to check that segment lifetime has not expired by comparing segment expiration time (i.e., sum of the timestamp and lifetime field) against its current time. Such a scenario is illustrated in figure 4.1. To change "could potentially resolve" at the beginning of this paragraph into "could actually resolve" the clocks of the sender and receiver must be synchronized.

Clocks are said to be synchronized when they run at the same frequency and their time is set so that they agree at a particular epoch with respect to coordinated universal time (UTC) [9]. Fortunately, the challenge of synchronizing computer clocks has already been solved by Network Time Protocol (NTP) [7]. NTP is a distributed system consisting of primary and secondary time servers, clients, and interconnecting transmission paths [9].

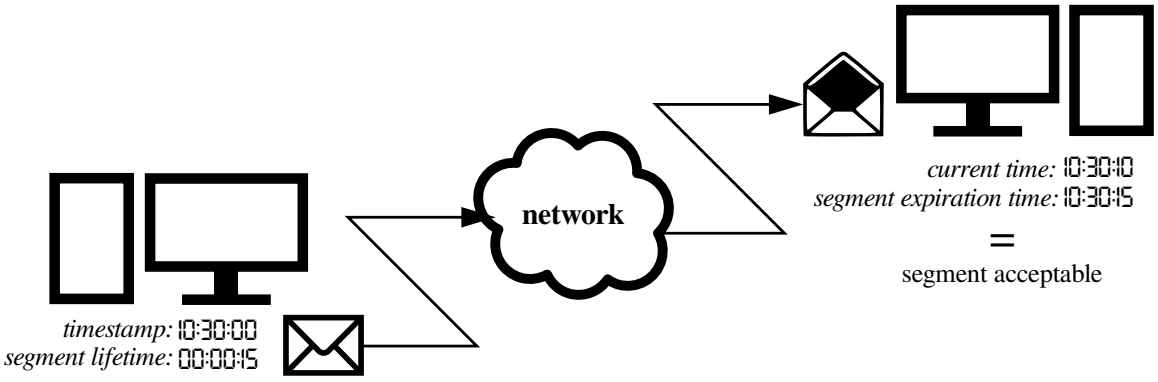

Figure 4.1: Bounding segment lifetime using timestamp and lifetime field.

The objective of NTP is to synchronize the clocks in all participating computers to the order of less than a millisecond or two relative to UTC [10]. The ins and outs of NTP protocol are not relevant to this work.

NTP daemon is a software implementation of NTP protocol [10]. It is readily available for any Linux distribution. Having NTP daemon setup and running is the prerequisite for correct Delta-t operation[1]. In the rest of this work, we will assume that clocks of communicating computers are synchronized.

4.1.2 Recovery Intervals

In its normal mode of operation Delta-t connection should not be closed until both Stimer and Rtimer expire. Otherwise, connection management hazards will reappear once again. Recovery interval must be in place to face computer crashes (or even ordinary reboots). They ensure that sender or receiver wait long enough before initiating or accepting a connection to prevent these hazards.

It has been decided that the current implementation of Delta-t will not take recovery intervals into account. The problem is that Linux kernel does not store the information about open sockets in persistent memory. Doing so would be very inconvenient and would never be bulletproof. For example, the computer could crash before the I/O operation has finished. As a result, recovery intervals cannot be enforced on individual sockets. It would be necessary to apply them at the very start of every connection (i.e., right after a socket has been opened). In consequence, first read operation would be delayed by Δt and first write by $3 \cdot \Delta t$.

In this work, no delay will be enforced at the very beginning of a communication after crash or reboot. Protocol implementation will be tested and evaluated in a setup where connection management hazards, resulting from the omission of recovery intervals, are guaranteed to be nonexistent.

4.1.3 Interaction with Socket API

Delta-t was originally designed to reliably transmit stream of messages. Message boundaries were marked in segment header. This type of communication is in contemporary socket API represented by `SOCK_SEQPACKET` socket type. In this work, Delta-t protocol will be

[1] As a consequence, Delta-t is dependent on the UDP protocol because NTP packets are carried by UDP datagrams [10].

implemented as if it was a byte-stream oriented protocol. As a result, Delta-t sockets will be of type `SOCK_STREAM`.

The intention is to make Delta-t behave the same way as TCP does when elementary socket functions are invoked. Therefore, the sequence of socket function calls performed by Delta-t to communicate with other party will be the same as of TCP. Porting a TCP application to Delta-t will be only the matter of changing the last argument of the initial `socket()` function call (i.e., `socket(AF_INET, SOCK_STREAM, IPPROTO_DELTAT)`). Also, this decision will make the process of testing and evaluating the implemented protocol less of a burden.

Typical Delta-t client–server style of a communication, in relation to elementary socket functions, is depicted in figure 4.2. In this diagram two differences can be seen when compared to TCP.

Executing `connect()` by the client will not result in Delta-t segment being transmitted. The connection will not be established until the first data segment is sent (by means of `write()`/ `sendmsg()`, or any related system call).

Furthermore, there is no connection termination signalling between the client and the server. The socket stays open until explicitly closed by the `close()` system call (assuming all segments are being successfully delivered between the two). Even when timers are not running (i.e., enough time has passed since last receive or send operation) the socket stays open. It is application's responsibility to close the socket when appropriate.

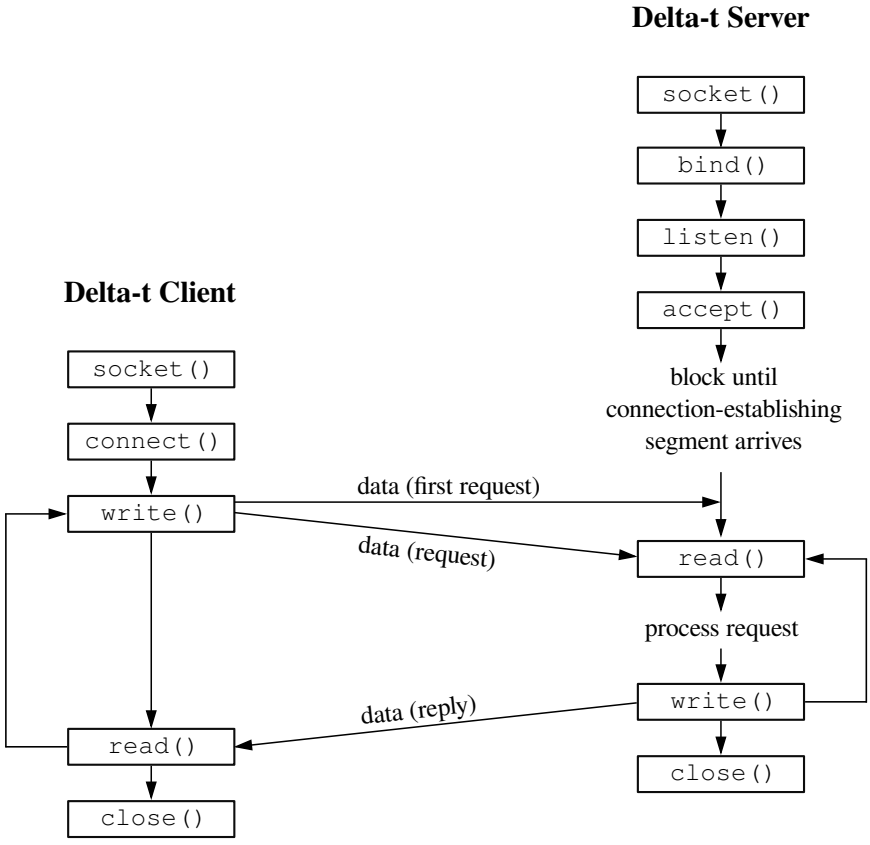

Figure 4.2: Socket functions for elementary Delta-t client–server.

4.1.4 Delta-t state diagram

State transition diagram depicts the operation of a protocol regarding connection establishment and connection termination. Transitions from one state to another are based on the current state and the event that occurred. [14]

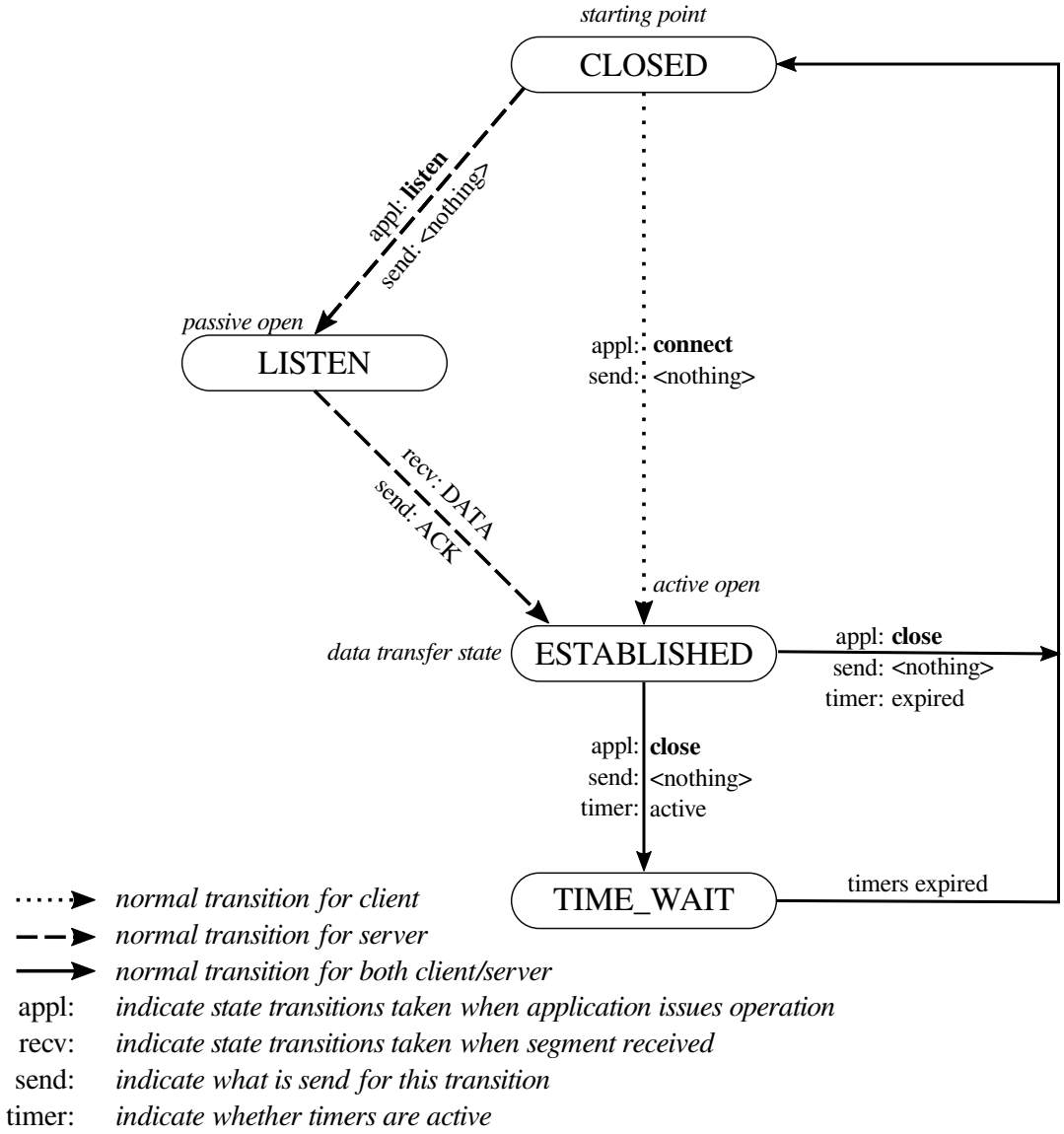

Figure 4.3: Delta-t state transition diagram.

Based on Delta-t protocol description, diagram depicted in figure 4.3 has been proposed. Socket is initially in the `CLOSED` state. By issuing the `listen()` system call, it transitions to the `LISTEN` state. This kind of opening is commonly referred to as *passive open*. It is performed by server application. Socket will remain in the `LISTEN` state until an acceptable segment is received. It could be either rendezvous or data segment. *Data-run* flag must be set in this segment. When this segment arrives the connection is established, i.e., the socket

moves to the ESTABLISHED state. Internally, the accept()[2] socket function has to be called to establish the connection. Immediately after server socket gets to the ESTABLISHED state its Rtimer is initialized because a segment was just received. When the server decides to send response to the client, its Stimer will be initialized as well.

Now, let us describe how the connection is established from client's perspective. Client must first call connect(). Unlike TCP, no network traffic will be generated. This call should only register peer's address. As a result, connect() will return immediately[3] and will always succeed (if valid address structure was given). The connection is considered to be established from now on, i.e., the socket transitions to the ESTABLISHED state. Initiating a connection as a client is referred to as *active open*. Client's timers are initialized when first data segment is sent or received.

In theory, there is third possibility to open a connection, which is not present in TCP. Two clients can make a connection between themselves. However, they need to know their corresponding addresses in order to pass them to the connect() function. Once they reach the ESTABLISHED state, they are free to exchange data.

From the ESTABLISHED state onward, state transitions are the same for both client and server. The socket is kept in this state even when timers expire. It is up to the application to decide when to close the socket with the close() system call.

If close() is invoked the socket moves either back to the CLOSED state, or transitions to the TIME_WAIT state, depending on the state of timers. In case Rtimer and Stimer are not set the socket moves back to the CLOSED state. Subsequently, any resource that has been allocated can be freed.

If at least one of the timers is running the path to the TIME_WAIT state must be taken. The socket will stay in this state until both expire. The rationale behind the TIME_WAIT state is the following. Being in the TIME_WAIT state means that the socket was closed prematurely. Ideally, an application should not be able to close the socket when timers are running. Unfortunately, this rule cannot be enforced in socket API. Applications are free to close the socket at any time. Using the TIME_WAIT state we keep the socket open at protocol level. With this approach, we stick to Delta-t specification – the socket returns to its default state when both timers go to zero. All segments, received in the TIME_WAIT state, are discarded because we know the data can never reach user application (user-level socket is already closed). Local socket address (i.e., local IP address and port) of a socket in the TIME_WAIT state cannot be reused. Trying to bind a new socket to this address will fail. SO_REUSEADDR socket option can be used to circumvent this behavior. Nevertheless, it is not advised as it might influence protocol reliability.

4.1.5 Header Format

In general, Delta-t kernel module should provide reliable delivery of a stream of bytes[4]. Additionally, full-duplex mode of operation should be supported and the communication should be flow controlled. Delta-t segment header will be constructed based on these criteria.

[2] accept() is a blocking call, servers usually call accept() right after listen() and wait for incoming connections.

[3] connect() in TCP returns only after 3WHS is finished or an error occurs [14].

[4] See subsection 4.1.3 for an explanation why a stream of bytes was chosen instead of a stream of messages.

UDP datagram header forms the basis of Delta-t segment header. Consequently, valid Delta-t segments can be considered as valid UDP datagrams. Also, there is an opportunity to implement Delta-t as user-space library, using UDP to transport Delta-t segments.

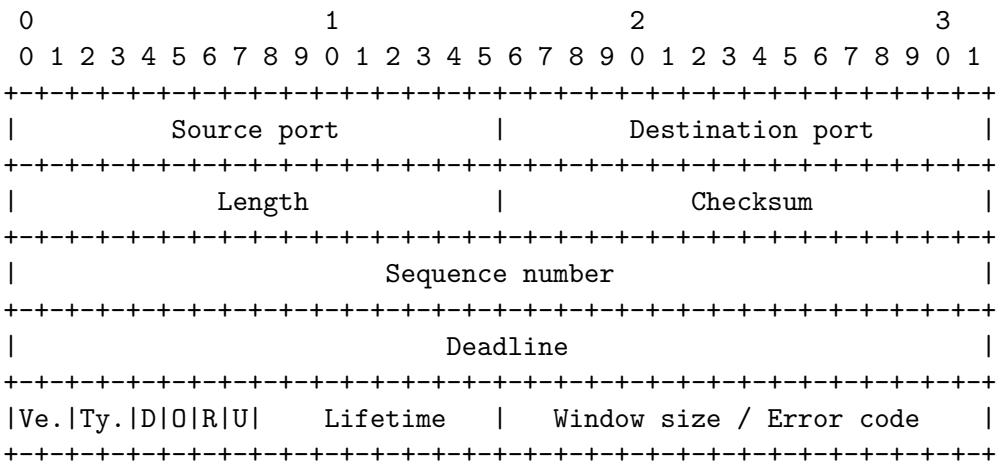

Figure 4.4: Delta-t segment header format.

The format of Delta-t segment header is illustrated in figure 4.4. There are no optional fields so the header is always twenty bytes long. At the segment level, Delta-t can support only half-duplex communication because there is only one field to store the SN. Unlike TCP, Delta-t segments carry either data or an ack, never both. However, full-duplex mode of communication is supported at the protocol level. In the following text, description of individual header fields will be given:

- `Source port` (16 bits) — Delta-t source port number.
- `Destination port` (16 bits) — Delta-t destination port number.
- `Length` (16 bits) — length of this segment, including Delta-t header, in octets (the minimum value of this field is 20).
- `Checksum` (16 bits) — standard UDP checksum.
- `Sequence number` (32 bits) — based on the segment type this field contains:
  - data segment — the SN of the first data octet in this segment.
  - rendezvous segment — the SN of the ack segment that announced zero window.
  - ack segment — the next SN the receiver is expecting in next data segment.
  - nak segment — the SN of the segment that caused the generation of this nak.
- `Deadline` (32 bits) — Delta-t segment expiration time. It has the same format as timestamp field of IP timestamp option – a right-justified, 32-bit timestamp in milliseconds modulo 24 hours from midnight UTC [15].
- `Version` (2 bits) — Delta-t version number (currently always set to zero).
- `Type` (2 bits) — Delta-t segment type. One of the following:

  - `0`: Data segment.
  - `1`: Ack segment.
  - `2`: Rendezvous segment.
  - `3`: Nak segment.

- Flags (4 bits)
  - `D`: *data-run* flag, set in data/rendezvous segment, if all previously sent segments have been acknowledged.
  - `O`: *overflow* flag, set in ack segment, when window overrun occurs.
  - `R`: *reliable* flag, set in ack segment, when zero window opens.
  - `U`: unused bit, always set to 0.

- `Lifetime` (8 bits) — lifetime of this segment, in seconds. It is the value of Δt. When Rtimer is not set and first acceptable segment is received, receiver's Rtimer is initialized based on the value of this field (i.e., Rtimer is set to $2 \cdot \Delta t$).

- `Window size/Error code` (16 bits)
  - data/ack/rendezvous segment — the number of data octets which the sender of this segment is willing to accept.
  - nak segment — the error code.

4.2 Implementation

When developing new protocol for the Linux kernel, it is much easier to use existing protocols as a reference. Also, using fragments of code that has already proven to work makes the development process faster, rather than starting from scratch. [8]

In this work TCP will be used as a reference. The intention is to follow TCP as close as possible so that anyone acquainted with Linux TCP stack will find the code in Delta-t module easy to understand. At the same time, anyone who will be able to grasp Delta-t implementation will also gain an insight into the Linux TCP stack.

The main goal of this work is to make a proof-of-concept implementation of Delta-t. With respect to this fact, we will target only IPv4 and leave out any additions that are required for IPv6 support. Moreover, Linux is constantly evolving with new releases being published fairly frequently. Delta-t implementation presented in this work is based on the version 5.11 of Linux kernel released on 14th of February 2021. Porting to older or newer kernels in the 5.0+ kernel series should not require much work, however.

There is a quite a bit of complexity associated with reliable transport protocols. In the rest of this chapter we will point only the key components and aspects of Delta-t implementation. There might not always be a perfect match between the code in Delta-t module and code snippets found in the following subsections. Some snippets were simplified in order to make them shorter or to give an explanation more easily.

4.2.1 Protocol registration

Transport layer protocols must have an internet protocol identifier/number assigned. This identifier is placed in the *protocol* field of IPv4 header (*next header* field of IPv6 header).

For example, 6 is the protocol number of TCP and 17 is the protocol number of UDP. Number 253 has been chosen as Delta-t protocol identifier. It is perfect fit for our purpose because it is meant to be used for experimentation and testing[5].

When registering new transport layer protocol, we need to tell the kernel what the protocol number is and what handler it should use when a network packet is received. For `AF_INET` protocol family it is accomplished by the function `inet_add_protocol()`. It takes `struct net_protocol` and protocol number as arguments. Figure 4.5 shows the invocation of `inet_add_protocol()` alongside the definition of its arguments. After this portion of code is executed, the `deltat_v4_rcv()` function will be invoked for every Delta-t segment received (identified by number 253 in corresponding IP header field). Callback `deltat_v4_err()` is responsible for processing ICMP packets that have Delta-t segment as a payload.

```
#define IPPROTO_DELTAT 253
static const struct net_protocol deltat_v4_protocol = {
        .handler        = deltat_v4_rcv,
        .err_handler    = deltat_v4_err,
        .no_policy      = 1,
        .netns_ok       = 1
};
inet_add_protocol(&deltat_v4_protocol, IPPROTO_DELTAT);
```

Figure 4.5: Adding input handlers for Delta-t protocol.

Although the kernel gained the ability to process Delta-t segments, it will eventually discard every single one of them. It will try to find a socket associated with a given segment based on IP addresses and port numbers, but no socket will be found, and the segment will be dropped. No socket will be found because there can be none. We have not told the kernel how to create Delta-t sockets yet (or even that Delta-t protocol exists). To register Delta-t protocol with the kernel `inet_register_protosw()` must be used. In figure 4.6 we can see kernel representation of a protocol (i.e., `struct inet_protosw`) and subsequent registration.

```
static struct inet_protosw deltat_protosw = {
        .type       = SOCK_STREAM,
        .protocol   = IPPROTO_DELTAT,
        .prot       = &deltat_prot,
        .ops        = &inet_deltat_ops,
        .flags      = INET_PROTOSW_ICSK
};
inet_register_protosw(&deltat_protosw);
```

Figure 4.6: Registration of Delta-t protocol with the kernel.

[5]See https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

Linux kernel keeps supported protocols in linked lists. There is a separate list for each protocol type (one for `SOCK_STREAM`, one for `SOCK_DGRAM`, etc.). Function call in figure 4.6 extends one of these lists by adding `deltat_protosw`. When Delta-t socket is opened by `socket(AF_INET, SOCK_STREAM, IPPROTO_DELTAT)`, list of `SOCK_STREAM` protocols is traversed until some of them matches `IPPROTO_DELTAT`.

Simple `socket()` function call opens three sockets. User-level socket, its internal kernel representation – `struct socket` and associated NET socket – `struct sock`. For Delta-t sockets the `ops` field of `struct socket` will point to `inet_deltat_ops` and the `sk_prot` field of `struct sock` will point to `deltat_prot`.

Every function (except `deltat_v4_rcv()` and `deltat_v4_err()`) in Delta-t module can characterized as follows: it is either a callback function used to initialize `deltat_prot` or `inet_deltat_ops`, or it is a utility function invoked by a callback.

`inet_deltat_ops`

Structure `inet_deltat_ops` contains 17 callbacks in total. Only three of them, shown in figure 4.7, were reimplemented while the rest (omitted for the sake of brevity) is shared with other transport layer protocols.

Moreover, these three functions are almost identical to their TCP counterparts with only minor modifications being made. For example, `connect()` callback in TCP first calls protocol specific[6] connect function (i.e., `tcp_connect()`) and then blocks waiting for the 3WHS to complete. `deltat_inet_connect()` does the same thing except it does not block and returns as soon as the protocol specific function is done.

```
const struct proto_ops inet_deltat_ops = {
        ...
        .connect        = deltat_inet_connect,
        ...
        .poll           = deltat_poll,
        ...
        .listen         = deltat_inet_listen,
        ...
};
```

Figure 4.7: Initialization of `inet_deltat_ops`.

`deltat_prot`

Compared to `inet_deltat_ops`, initializing `deltat_prot` is much more involved. At first, having both `inet_deltat_ops` and `deltat_prot` may seem redundant. However, this is not the case. `inet_deltat_ops` is oriented towards the BSD socket while `deltat_prot` is interfacing with the network. As a result, all the networking-related handlers as well as other information is kept in `deltat_prot`. This structure has a lot of members. Most of them are defined in Delta-t module (i.e., definitions from other protocols could not be reused).

[6]The mechanism of invoking protocol specific handler has been discussed in subsection 3.2.1, where `struct sock` data structure is described.

In figure 4.8 few notable members of `deltat_prot` can be seen with their corresponding values. Description of each member is given in the following text.

```
#define MAX_DELTAT_HEADER (sizeof(struct deltathdr) + 64 + MAX_HEADER)
static struct proto deltat_prot = {
        .connect                = deltat_v4_connect,
        .sendmsg                = deltat_sendmsg,
        .backlog_rcv            = deltat_v4_do_rcv,
        .h.hashinfo             = &deltat_hashinfo,
        .hash                   = inet_hash,
        .max_header             = MAX_DELTAT_HEADER,
        .obj_size               = sizeof(struct deltat_sock),
        ...
};
```

Figure 4.8: `deltat_prot` (incomplete) initialization.

- `connect()` — In contrast with TCP, Delta-t specific connect procedure, which is `deltat_v4_connect()` (called by `deltat_inet_connect`), only consults the destination address with IP layer and records it for subsequent use.
- `sendmsg()` — All write operations on a BSD socket (i.e., `write()`, `send()`, `sendmsg()`, etc.) will eventually invoke this callback.
- `backlog_rcv()` — It is necessary to lock a socket before any user-level read operation so that the kernel cannot modify the underlying data. If a segment arrives in the meantime, it cannot be processed because the socket is locked. The segment is added to the backlog queue and processed by a callback saved in this field when the socket lock is released. Output operations lock the socket as well.
- `h.hashinfo` — Transport protocols store bound sockets (i.e., sockets with IP address and port assigned) in a hash table. This hash table is consulted whenever new segment arrives to find corresponding socket.
- `hash()` – One of the few functions that could be reused. It adds a socket to the hash table just mentioned.
- `max_header` — When new Delta-t segment is allocated enough headroom must reserved in the data are of a corresponding SKB. Transport, network, and link layer headers must fit in this space. At each layer maximum header size is assumed and their sum is put in `max_header` field.
- `obj_size` — Transport layer protocols have to keep per socket state variables. For example, a timestamp of last segment sent or a SN that is expected to be received. Protocols extend NET socket (`struct sock`) by adding additional fields to it.[7] Naturally, the resulting data structure will be bigger than `struck sock`. When the extended NET socket is allocated `obj_size` is used to make space for additional state variables.

[7]The principle applied to accomplish this will be shown in subsection 4.2.2.

4.2.2 Data structures

Linux kernel uses simple scheme to extend existing data structures that mimics inheritance pattern from object-oriented languages. New `struct` has to be defined and the existing one must be placed at the very beginning. Subsequently, we can add in any additional fields.

The base structure is placed at the beginning to enable pointer casting. Using pointer casting we can access fields of the base structure more easily, especially when data structure is expanded multiple times.

Different protocols have different needs and the NET socket structure is not designed to cover all of them. Transport layer protocol has to extend `struct sock`[8] and make a custom version of it.

Network layer representation of a Delta-t BSD socket is `struct deltat_sock`. From now on the term "Delta-t NET socket" will refer to this structure[9]. Its full definition can be seen in figure 4.9. The definition is so short because it uses TCP NET socket as a starting point. Consequently, opening Delta-t socket will consume (slightly) more resources than opening TCP socket. Generally, Delta-t implementation presented in this work will not be better than TCP in terms of memory usage.

```
struct deltat_sock {
    struct tcp_sock tcp;
    u8              rcv_deltat;   // Δt for acks (not reliable-acks)
    u8              snd_deltat;   // Δt for all other segments
    u32             rcv_timer;    // 2 * rcv_deltat * HZ
    u32             snd_timer;    // 3 * snd_deltat * HZ
    u32             last_rcv;     // last new SN received timestamp
    u32             last_snd;     // last new SN sent timestamp
    u32             rtx_timeout;  // the value of R in Δt = MPL + R + A
    u8              rendezvous_state:3, // current rendezvous state
                    wnd_oflow:1,  // window overrun occurred
                    unused:4;
};
```

Figure 4.9: Definition of `deltat_sock` – network layer representation of Delta-t socket.

However, using TCP NET socket as a basis of Delta-t NET socket gives us the ability to reuse substantial portion of TCP logic. Some of it can be reused directly by a plain function call. This is possible when the function in question is explicitly exported. Unfortunately, most TCP functions are not visible from loadable modules. Instead of rebuilding the kernel, these functions were simply copied into the Delta-t module.

[8] To be precise, transport layer protocols do not extend `struct sock` directly. Instead, they use already expanded version of it. For example, custom TCP NET socket is `tcp_sock` and it uses `inet_connection_sock` as a basis. The complete sequence is the following: `tcp_sock` → `inet_connection_sock` → `inet_sock` → `sock` → `sock_common` (`struct sock` is an expansion of `struct sock_common`).

[9] Similarly, TCP NET socket will refer to `struct tcp_sock`.

4.2.3 Input Path

When looking at the implementation of Delta-t input handler – `deltat_v4_rcv()`, one can get lost fairly quickly. After initial sanity checks the code gets somewhat convoluted. In this subsection, we will describe input processing done for the first segment of a connection. We assume that the segment is valid and there exists a listening socket that will accept this segment.

Diagram in figure 4.10 shows the input path by means of a sequence of function calls. At each level in the hierarchy only the most important functions are depicted. Moreover, some of the functions are called using callback mechanism, but in the diagram, they are presented as if they were called directly.

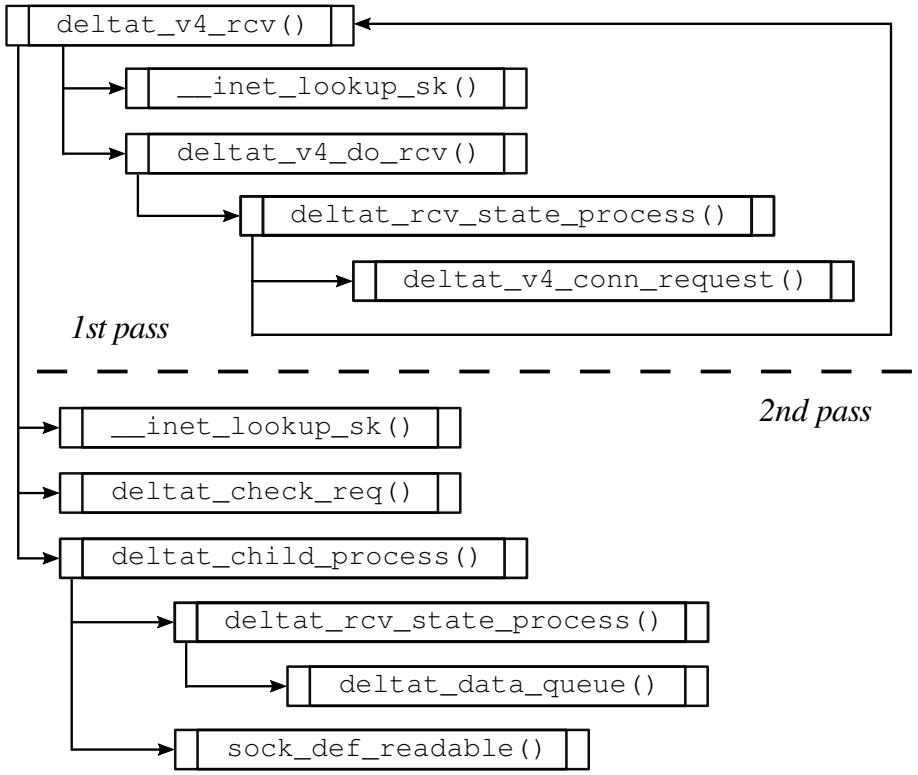

Figure 4.10: Path taken by the first (valid) data segment received by a listening socket.

The primary input handler is actually called twice for each connection-establishing segment (i.e, first data segment with *data-run* flag set). After segment validity has been tested, `__inet_lookup_sk()` is called. In our scenario, it will find the associated listening socket. `deltat_v4_do_rcv()` is used to quickly demultiplex segments for established sockets. However, our socket is in the `LISTEN` state. `deltat_rcv_state_process()` handles segments of non-established sockets. `deltat_v4_conn_request()` checks that the queue of requests is not full and creates new request socket. Request socket is minimalistic version of Delta-t NET socket. To create fully formed Delta-t NET socket a (recursive) call to `deltat_v4_rcv()` is made. This time around, `__inet_lookup_sk()` will return the request socket instead of the listening one. Delta-t NET socket is allocated and initialized by `deltat_check_req()`. Subsequently, `deltat_rcv_state_process()` will be called

again. New socket will be in the `SYN_RECV`[10] state and the segment will be added to the receive queue using `deltat_data_queue()`. Finally, the parent socket will be woken up by `sock_def_readable()`.[11]

Segments received in the future will be handled by `deltat_rcv_established()` (not in the diagram) and the path will less complicated: `deltat_v4_rcv()` → `deltat_v4_do_rcv()` → `deltat_rcv_established()`.

4.2.4 Output Path

Unfortunately, Delta-t output path cannot be described as a simple sequence of function calls. External events influence which segment will be transmitted at what time. All output operations will eventually invoke `deltat_transmit_skb()`. This function processes Delta-t SKBs based on the information stored in the control block (`cb[]` field of a SKB). It prepends Delta-t header to each segment and passes it to the network layer. Few possible ways to reach `deltat_transmit_skb()` are shown in figure 4.11.

The most obvious starting point is the `deltat_sendmsg()` function. It is invoked when a socket is written to. Less obvious one is `deltat_data_snd_check()`, which is called at the end of input processing just before `deltat_v4_rcv()` returns.[12]

Output operation can be initiated by `deltat_retransmit_timer()`. As the name suggests, it is responsible for handling retransmission timer events. Unlike the previous two functions, this one does not use `deltat_write_xmit()` along the way. The reason for it is quite simple. Segments to retransmit are in the retransmit queue, not in the send buffer where `deltat_write_xmit()` expects them.

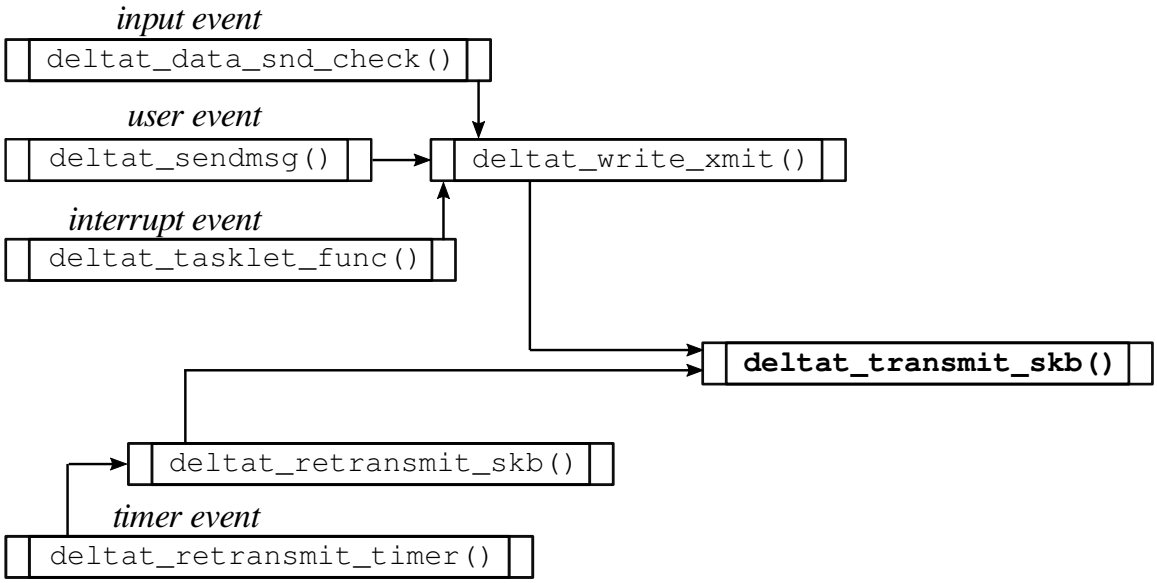

Figure 4.11: Incomplete list of Delta-t output functions and their interrelation.

[10] This is purely internal state indicating that the socket was just allocated. The socket will transition to the `ESTABLISHED` immediately after some more initializations are done.

[11] The parent socked is probably sleeping in the `accept()` system call.

[12] `deltat_data_snd_check()` was omitted from the discussion in the previous subsection for the sake of brevity.

There can be only limited amount of SKBs in the networking queues below Delta-t at any instant of time. If they are full, Delta-t must delay the sending of new SKBs. Once a segment leaves the network device, destructor callback will be invoked to free the corresponding SKB. That callback will schedule a task (`deltat_tasklet_func()`) that will try to send new Delta-t segments by the time next software interrupt occurs. Destructor cannot issue the output operation itself because of locking.

There are two queues that hold outgoing segments. New segments are appended to the end of the send queue by `deltat_sendmsg()`. This queue is implemented as a doubly linked list and `sk_write_queue` field points to the head. Each SKB transmitted by Delta-t is removed from the head of the send queue and put in the retransmit queue. Retransmit queue is represented by `tcp_rtx_queue` field.[13] SKB is kept in the retransmit queue until acked. An illustration of send and retransmit queue is provided in figure 4.12. Initially, a) three SKBs are waiting in the send queue, retransmit queue is empty, and b) first segment (head of the send queue) is transmitted and placed in the retransmit queue.

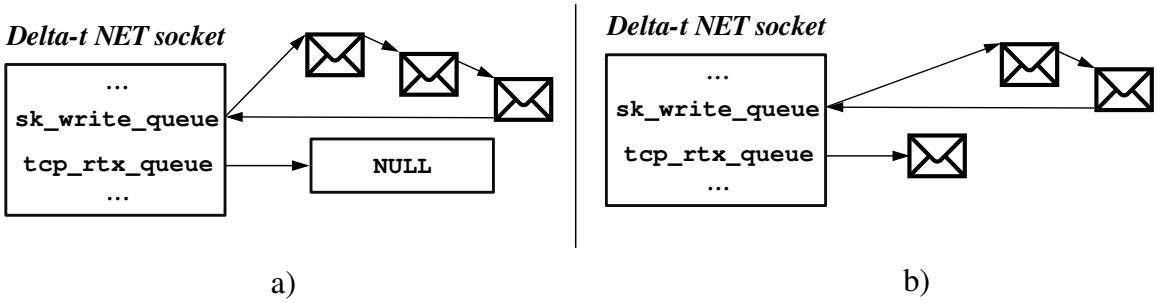

Figure 4.12: Output queues: a) before send operation b) after send operation.

In general, Delta-t output functions are very similar to their TCP counterparts but one aspect of Delta-t output differs significantly from TCP. Current implementation is not congestion controlled. Congestion-related variables are not updated, and congestion window quota is not respected. There is only one simple heuristic used. If no ack is received after three consecutive retransmits, we cease sending new data and focus only on transmitting segments from the retransmit queue. However, all other output logic (e.g., flow control, Nagle's algorithm, etc.) is kept in place.

4.2.5 Value of `Δt`

In fact, there are two `Δt`s, one for each direction of data flow. They are represented by `rcv_deltat` and `snd_deltat` in Delta-t NET socket.

`rcv_deltat` is initialized from `lifetime` header field when Rtimer is zero and first acceptable segment is received. This value is put in `lifetime` field of subsequent ack segments. `rcv_deltat` is controlled by the other end of a connection.

`snd_deltat` is initialized during socket creation and it reflects the value of global variable `deltat`, which also happens to be a parameter of the module. `deltat` is set to 32 seconds by default. This value was chosen so that the time spent in `TIME_WAIT` state is approximately the same compared to TCP[14] and there is enough time for retransmissions. The number of

[13] Unlike send queue, the underlying data structure of retransmit queue is as a red-black tree.

[14] In the worst-case scenario, Delta-t spends 1 minute and 36 seconds in the `TIME_WAIT` state. `TIME_WAIT` state of TCP is always 1 minute long (this holds for Linux).

retransmissions performed by Delta-t before giving up is dependent on the value of `deltat`. The retransmission logic will be discussed in the following subsection 4.2.6.

The value of `deltat` can be modified when the module is loaded into the kernel. The following snippet shows how to increase its default value (32 seconds) to 128 seconds.

```
sudo insmod deltat.ko deltat=128
```

Current `deltat` value is stored in `/sys/module/deltat/parameters/deltat`. To read it we can issue the following command.

```
cat /sys/module/deltat/parameters/deltat
```

The value can be tweaked even after the module has been loaded. The way to do it is shown in the snippet below. Executing this command will not affect existing sockets, however. Only sockets opened after this point will reflect the new value.

```
echo 16 | sudo tee /sys/module/deltat/parameters/deltat
```

Lifetime check

Segment expiration time is checked by the function `remaining_lifetime()`. Value in the `deadline` header field stores the number of milliseconds since midnight UTC. If a segment is sent less than `Δt` seconds before midnight, then its `deadline` value will wrap around. The `remaining_lifetime()` function takes wraparound into account and the segment should be correctly validated at any time.

4.2.6 Retransmission logic

Lifetime of Delta-t segments is bound. In consequence, there is no point in trying to retransmit segment that is already dead. Once this situation occurs, the socket is closed, and error is reported to the user application.

The same retransmission scheme that is used in TCP is followed. The retransmission timeout (RTO) is doubled with each retransmission – this is called *exponential backoff*. Currently, segment is being actively retransmitted for three quarters of its lifetime (24 seconds by default). The other half is spent waiting for an ack to arrive or retransmitting other segments that have not hit their maximum retransmission time. New segments are not sent in this period to meet rule R2 from section 2.1.

Ideally, RTO should not be too short so that segments are not retransmitted unnecessarily. On the other hand, it should not be too long either. Long delay before lost segment is retransmitted would have negative effect on performance. [16]

Delta-t uses different value to initialize the RTO than TCP because the subsequent behavior differs. TCP implements the algorithm from RFC 6298 – it takes round-trip time samples and updates the RTO value accordingly [11]. Delta-t implementation presented in this work does not take any round-trip time samples and the value is updated only by *exponential backoff*. RTO is set to 0.5 seconds during socket initialization[15]. After successful retransmission it is reset back to 0.5 seconds.

[15] TCP initializes RTO to 1 second.

Not only the retransmission timer can trigger retransmit event. Delta-t module, after having received three duplicate acks, performs fast retransmit. As a result, segment at the head of the retransmit queue is sent immediately, instead of waiting for the retransmit timer to expire.

4.2.7 Timer management

Timers are fundamental to proper Delta-t protocol operation. In Linux kernel, the flow of time can be measured by two types of timers. Both allow us to defer work to some point in the future. Generally, these two types differ by the resolution they have and by the precision they can achieve.

To implement Stimer/Rtimer logic no actual timer has been employed. This is possible because there is no work to be done associated with the timer expiration. In Delta-t module, instead of setting up timers we simply record the time of last *send* and *receive event*. These events are defined as follows:

- *Send event* is triggered whenever new data, rendezvous or reliable-ack is sent. Retransmissions of the previous segments are not considered as *send events*. The time of last *send event* is stored in `last_snd` field of Delta-t NET socket.
- *Receive event* is triggered whenever new data or rendezvous segment is received and accepted, or it is received and rejected because of window overrun. The time of last *receive event* is stored in `last_rcv` field of Delta-t NET socket.[16]

When new segment is received, we compare the elapsed time[17] against the value of $2{\cdot}\Delta t$[18]. If the elapsed time is bigger than $2{\cdot}\Delta t$, then any SN is acceptable and the SN stored inside the segment does not have to be checked. Thus, the segment will be accepted and the `last_rcv` field will be populated with current time because *receive event* just occurred.

In this implementation, we check the expiration of Stimer (i.e., comparing the elapsed time since last *send event* against $3{\cdot}\Delta t$) only on socket close. Expiration of Stimer indicates that the receiver will accept any initial SN. However, we decided to simply leave the SN unchanged throughout the lifespan of a socket. This unchanged value should be equally valid as any other value.

4.2.8 *Rendezvous-at-sender*

Receiving zero window advertisement (i.e., ack segment with zero window size) causes the *rendezvous-at-sender* procedure to be initiated. This procedure avoids periodic polling utilized in TCP. It is simple sequential process consisting of few steps. Current state in the process is held in the `rendezvous_state` field of Delta-t NET socket. *Rendezvous-at-sender* implementation presented in this work follows the diagram in figure 4.13.

`RENDEZVOUS NONE` is the default state where traditional data exchange takes place. However, as soon as an ack with zero window is received, sender sends a rendezvous segment as a response and transitions to the `RENDEZVOUS SENT` state. Rendezvous segment must be acked. Doing so changes the state to `RENDEZVOUS ACKED`. Sender will remain in this state until the window opens (i.e., until reliable-ack arrives).

[16] There are very similar fields to `last_snd` and `last_rcv` in the TCP NET socket, but they cannot be repurposed for Delta-t's use case.

[17] Calculated by subtracting the time of last *receive event* from current time.

[18] Which is precomputed and stored in the `rcv_timer` field of Delta-t NET socket (in jiffies).

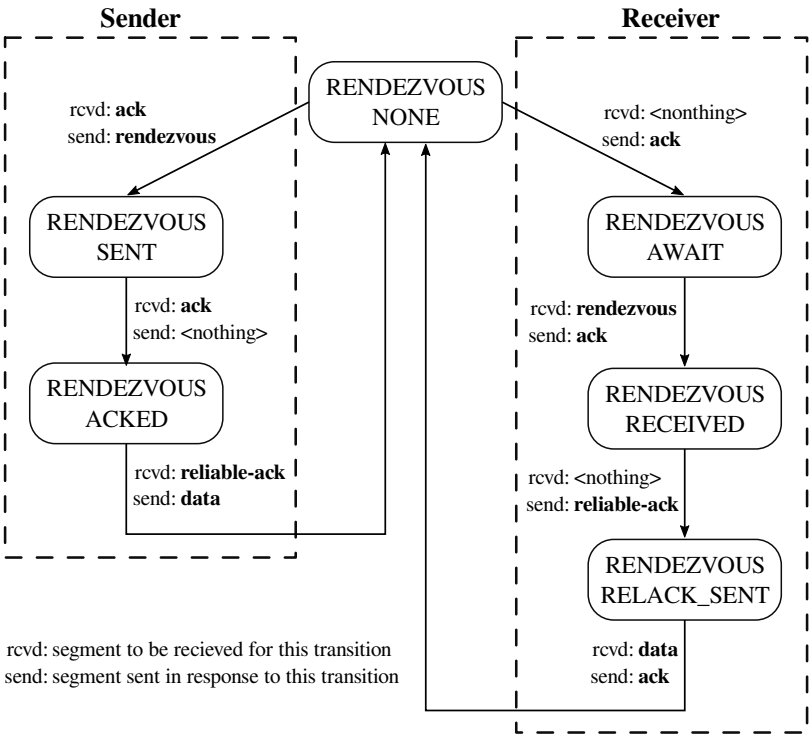

Figure 4.13: State diagram of the *rendezvous-at-sender* procedure.

Let us look at the previous scenario from receiver's point of view. When receive buffer is full and consequently receive window shrinks to zero, receiver marks this fact in the next ack to be sent[19]. Once this ack is sent, receiver will move to the `RENDEZVOUS AWAIT` state. Receiving rendezvous segment in this state has three outcomes. An ack will be generated for the rendezvous segment. Receiver's window will be shifted and state transition to the `RENDEZVOUS RECEIVED` state will be taken. Receiver can leave this state only when user application reads enough data from the receive buffer.

In contrast with the description of *rendezvous-at-sender* procedure given in section 2.2, rendezvous segments in this implementation of Delta-t do not carry the offset[20] value. Instead, the same fixed value is added to the SNs currently in use to shift the windows. Maximum window size (i.e., 65 535[21]) is used as offset. It is assumed that the sender cannot send more than a full window of data.

4.2.9 Utilities

Packet analyzers are very handy tools when troubleshooting network related issues. Even more so when new transport protocol is being implemented. The ability to capture and analyze Delta-t traffic was essential throughout the development process.

Probably the most popular packet analyzers are `wireshark` (GUI application) and `tcpdump` (CLI application). Both have been extended by adding Delta-t protocol support.

[19] If some data could not fit in the receive buffer the *overflow* flag will be set in this ack.

[20] As a remainder, offset is used to shift sender's and receiver's window so that any possible duplicates will not cause issues.

[21] `Window size` field is 16 bits long. Therefore, the value is $2^{16} - 1$.

Particularly, LUA dissector plugin has been written for `wireshark`. We can see wireshark in action in figure 4.14.

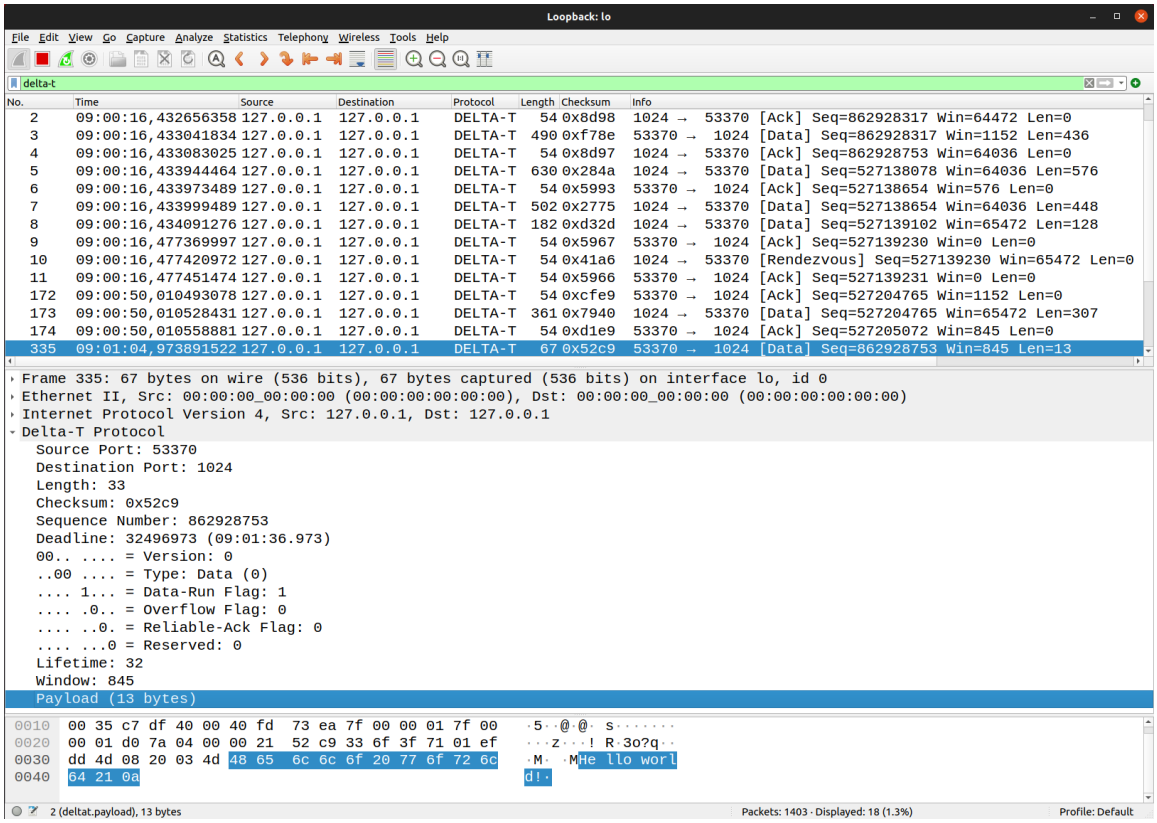

Figure 4.14: Using `wireshark` to analyze Delta-t segment.

Extending `tcpdump` was more involved because it does not have any plugin infrastructure. Modifications have been made directly to its source code. Fortunately, the required changes were straightforward. Nevertheless, to use `tcpdump` with Delta-t support, we must download its dependencies and recompile/build it.

4.2.10 Testing

For ad hoc testing, especially in the early stages of development, a simple echo client–server application was implemented. Client reads data from standard input line by line and then transmits each line to the server. Server echoes every received segment back to the client. However, it took considerable amount of time until the implementation was mature enough to act according to the scenario just mentioned.

Manual testing proved to be very useful when the module was not fully operational, but it became increasingly tedious once the module started working as expected. As a result, some other approach had to be devised, ideally a scriptable one. We decided to solve this issue by using `packetdrill` to test Delta-t implementation presented in this work.

`packetdrill` is an open source utility, released by Google, aimed at testing the entire TCP/UDP/IP network stack in a scriptable fashion. It defines its own scripting language to specify test scenarios. This language enables us to represent packet events, or to execute system calls and shell commands. Simply put, packet event is a description of inbound

or outbound packet. When an inbound packet event is encountered in the script file, real packet is constructed based on the description and injected into the kernel. When an outbound packet is sent by the kernel, it is sniffed by `packetdrill` and compared against the packet event provided in the script file. [1]

The source code of both, the packetdrill tool and the scripting language, has been modified to enable testing of Delta-t protocol. Individual test cases were written as new features were implemented. Currently, there are 17 tests. These tests simulate: simple communication scenarios, invalid socket function calls, out of order delivery, sending expired segments as well as the *rendezvous-at-sender* procedure. `packetdrill` provides a utility script that runs all these tests one after another. An excerpt of a `packetdrill` script that simulates *rendezvous-at-sender* is illustrated in figure 4.15. Full version of the script can be found in appendix B.

```
// Establish a connection.
   +0 < deltat data D 0:1000(1000) win 32792
   +0 accept(3, ..., ...) = 4
   +0 > deltat ack - 1000:1000(0)
// The following segment should fill receiver's window
   +0 < deltat data D 1000:1460(460) win 32792
   +0 > deltat ack - 1460:1460(0) win 0     // Zero window adv.
// Trying to squeeze in another segment
   +0 < deltat data - 1460:2460(1000) win 32792
   +0 > deltat ack 0 1460:1460(0)           // Not succesfull
   +0 < deltat rendezvous D 1460:1460(0) win 32792
   +0 > deltat ack - 1461:1461(0)
// Empty the receive buffer
   +0 read(4, ..., 1460) = 1460
   +0 > deltat ack R 66995:66995(0)         // Reliable-ack sent
// (Overflow) segment that dit not fit can be sent again
   +0 < deltat data D 66995:67995(1000) win 32792
   +0 > deltat ack - 67995:67995(0)
```

Figure 4.15: Portion of a script that simulates *rendezvous-at-sender* with window overrun.

Let us briefly describe the script in figure 4.15. `packetdrill` uses C/C++ style syntax for comments. There are only two system calls shown (`accept()` and `read()`). The rest of the script is composed of packet events. The `+0` at the beginning of each line means that the line will be executed right after the previous line without any delay. Inbound segments are constructed by `packetdrill`, and they are denoted by `<`. Outbound segments, denoted by `>`, are generated by the kernel as a response to the inbound segment. Individual fields of a packet event are broken down in figure 4.16.

```
time direction deltat type flags start-SN:end-SN (length) win window-size
  +0     <     deltat data   D     66995 : 67995  (1000)  win    32792
```

Figure 4.16: Break down of a Delta-t packet event.

Chapter 5

Evaluation

In this chapter, the implementation of Delta-t protocol will be evaluated in different network scenarios. Moreover, comparison with TCP, operating in the same conditions, will be provided. We will accomplish this task using two directly connected virtual machines.

In this work, we used `virtualbox` platform. Two guest machines were instantiated having 2 GB of available memory and running Ubuntu distribution kernel version 5.11. Guest machines are configured to use `virtualbox` internal networking mode. `Virtualbox` support driver acts as a Ethernet switch between the two. Additionally, both operating systems use `Intel PRO/1000 MT Desktop` virtualized networking hardware.

Over the years TCP has registered many improvements, which increase its overall performance. For example, RFC 4614 which keeps track of RFCs related to TCP, registers 26 of them as recommended enhancements [3]. Linux implements most of these enhancements.

However, we will attempt to provide unbiased comparison and disable performance-related TCP extensions. In Linux, the TCP stack can be tuned via `sysctl` parameters[1]. We will turn off the following features: TCP window scaling, receive buffer auto-tuning, selective acknowledgements (SACKs), duplicate SACK support and F-RTO algorithm. The command to disable all of these features at once is shown in figure 5.1.

```
sudo sysctl -w \
  net.ipv4.tcp_window_scaling=0 net.ipv4.tcp_moderate_rcvbuf=0 \
  net.ipv4.tcp_sack=0 net.ipv4.tcp_dsack=0 net.ipv4.tcp_frto=0
```

Figure 5.1: `sysctl` command to disable TCP advanced features.

Furthermore, TCP supports software and hardware offload mechanisms that can also improve its performance. In context of TCP, we are referring to: TCP segmentation offload (TSO), generic segmentation offload (GSO) and generic receive offload (GRO). Using these mechanisms, TCP can pass to or receive from the network layer segments that exceed the maximum segment size of the connection. These offloads are turned off by the command in figure 5.2.

The performance of the two protocols will be assessed using simple file transfer client–server application. `iperf` tool relies on the fact that (reliable) transport protocol explicitly

[1] These parameters are documented at https://www.kernel.org/doc/html/latest/networking/ip-sysctl.html.

```
sudo ethtool -K <interface> tso off gso off gro off
```

Figure 5.2: `ethtool` command to modify networking hardware settings.

closes the connection, but this is not the case for Delta-t. As a result, `iperf` could not be used for benchmarking because the results would be inaccurate.

The order of operation of the benchmarking file transfer application is straightforward. Client first opens a file, determines its size, sends the size to the server, and starts transferring the data. This way, the server knows how many bytes are expected (i.e., after how many bytes the file transfer will be over). The server measures and reports elapsed time of the file transfer. Subsequently, we will calculate the throughput based on the file size and the time of the file transfer. The throughput reported in the following sections is an average of 10 runs.

We will use Delta-t default lifetime (i.e., 32 seconds) and we will simulate the same setting in TCP through `TCP_USER_TIMEOUT` socket option. This option specifies the maximum amount of time data may stay buffered without being transmitted, or transmitted data may remain unacknowledged. After this period, the TCP connection will be closed. Snippet shown in figure 5.3 sets the client's user timeout to 32 seconds. Using this option, we will overwrite default behavior of TCP. By default, TCP in established state can send up to 15 retransmits before reporting a failure to the user application. Depending on the current RTO it may take from 13 to 30 minutes to send all these retransmits.

```
unsigned timeo = 32000;
setsockopt(sockfd, IPPROTO_TCP, TCP_USER_TIMEOUT, &timeo,
           sizeof(timeo));
```

Figure 5.3: Setting the TCP timeout option to 32 seconds.

Different network properties in the rest of this chapter are simulated using `netem` (network emulation) module. This module is controlled by `tc` command line utility. Using a simple notation, we can instruct a networking interface to delay, lose, damage, reorder, or duplicate packets. It is important to note that these events affect only outgoing packets. In each scenario only one interface will be controlled by `netem` and, as a result, only the traffic in one direction will be altered.

5.1 Scenario 1

This is the simplest scenario to be presented. We will study how network delay affects the protocol performance. More precisely we will take a closer look at the protocol operation with 50, 100, 200, 300, 400 and 500 milliseconds round trip time delays. The round trip time delay is set using the command in figure 5.5.

```
sudo tc qdisc add dev <interface> root netem delay <delay>ms
```

Figure 5.4: Command to instruct `netem` to simulate network delay.

The results are shown in the graph in figure 5.5. It can be seen, that TCP was more performant in each case. However, the performance gap between the two stays approximately the same. The performance gap (i.e., how much is Delta-t less performant than TCP) is shown in table 5.1.

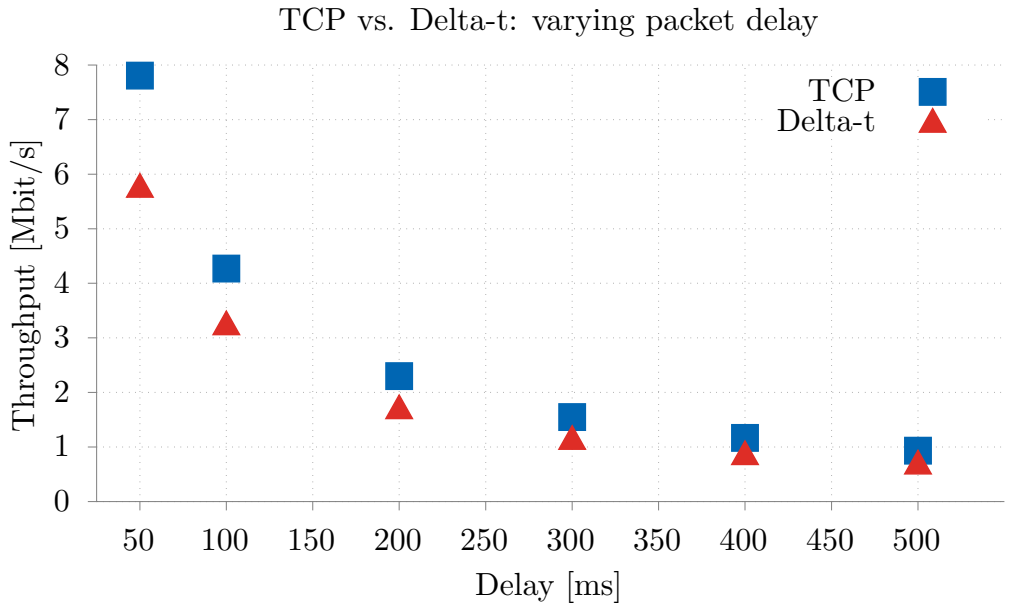

Figure 5.5: Test scenario with with varying packet delay.

Delay	50 ms	100 ms	200 ms	300 ms	400 ms	500 ms
performance gap	26,11 %	23,93 %	25,56 %	25,83 %	25,517 %	25,28 %

Table 5.1: The difference between Delta-t and TCP performance

5.2 Scenario 2

This scenario is similar to the previous one, except now we will introduce packet loss. The packet loss ratio will stay fixed at 5 % while the network delay will vary. Corresponding `netem` command is in figure 5.6. The results are summarized in the graph in figure 5.7.

```
sudo tc qdisc add dev <interface> root netem loss 5% delay <delay>ms
```

Figure 5.6: Command to instruct `netem` to simulate network delay and 5 % packet loss.

Delta-t was able to achieve higher throughput than TCP. This might be caused by TCP congestion control algorithm which slows down TCP output in the event of packet loss. In the next scenario, we will see that the absence of congestion control in Delta-t makes the protocol less reliable, nevertheless.

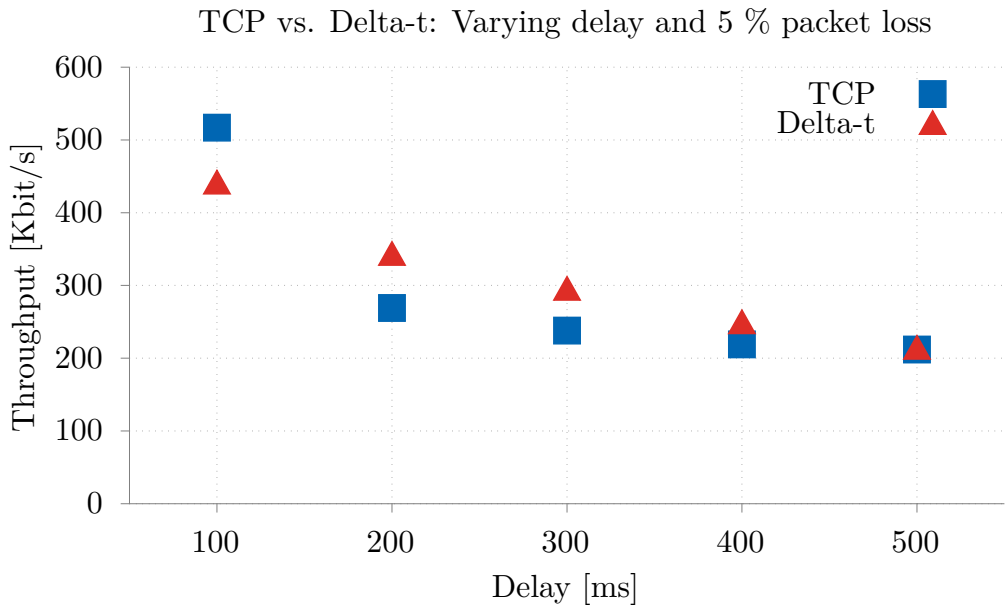

Figure 5.7: Test scenario with fixed packet loss (5 %) and varying packet delay.

5.3 Scenario 3

In this last scenario, we will increase the packet loss ratio and keep the delay fixed at 50 milliseconds. The impact on the performance is depicted in the graph in figure 5.8.

From the results we can see that TCP deals with packet losses much better than Delta-t. Additionally, Delta-t file transfer occasionally failed when packet loss reached 20 %. This failure is caused by the lack of congestion control algorithm. Delta-t sends segments in bursts. When this burst is sent all the segments are put in the retransmit queue and their lifetime starts counting down. Lost segment is retransmitted when third duplicate acks is received, or when the retransmission timer expires. Either way, only the head of the retransmit queue is retransmitted. With higher packet losses the time spent in the retransmit queue increases as well as the probability that the segment lifetime will expire, while being in the retransmit queue.

One solution to this problem would be to increase the segment lifetime – value of `deltat`. However, it would not solve the issue, only postpone it. Proper solution would be to implement congestion control mechanism. With this approach there would be a manageable number of segments in the retransmit queue because new segments would stay longer in the send queue where their lifetime is not counting down.

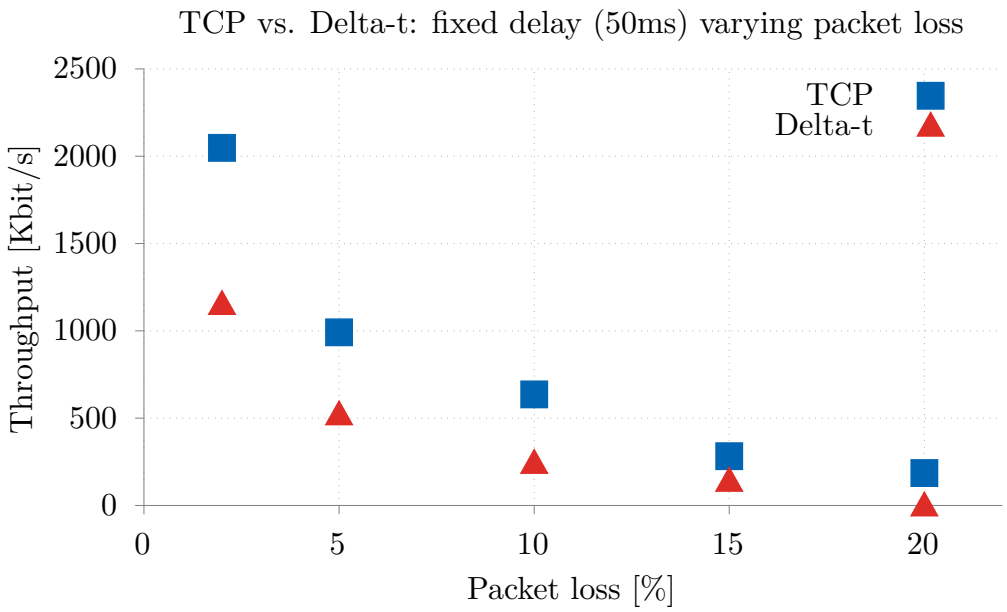

Figure 5.8: Test scenario with fixed packet delay (50 ms) and varying loss ratio.

Chapter 6

Conclusion

In this work we studied Delta-t protocol design and its major features. We learned that the protocol relies heavily on the packet aging services provided by lower layer. Moreover, a brief description of Linux networking subsystem and its ability to load separate binaries into the kernel image was provided.

Delta-t protocol cannot operate on top IP as is. First, we had to make some modifications to it, to make it compatible with IP and lay out the plan for the subsequent implementation. Luckily, substantial portion of TCP logic could be reused. Nevertheless, the implementation of any reliable transport layer protocol is very complex. Delta-is no different in this regard. In consequence, only the key aspects of the implementation were pointed out.

Finally, we provided performance comparison with TCP. Before starting with the evaluation, we first turned off TCP advanced features (i.e., offloads, SACKs, etc.) to make unbiased comparison. In summary, TCP is more performant and more reliable.

Current version of Delta-t module is not perfect. However, there is still a lot of room for improvement. Congestion control mechanism and better RTO estimations would definitely enhance Delta-t reliability and performance. Also, the memory usage of Delta-module could be significantly reduced, and better observability of the module operation, through a set of statistics, would be desirable as well.

To sum up, the area of reliable transport protocol implementation is very broad and complex and to implement one is not a one-man job. It took decades and several dozens RFCs for TCP to reach its current state. Nevertheless, the main goal of this work was not to make an alternative to TCP. Instead, this work is trying to meet the objective of making a proof-of-concept implementation of Delta-t.

Bibliography

[1] Cardwell, N., Cheng, Y., Brakmo, L., Mathis, M., Raghavan, B. et al. Packetdrill: Scriptable Network Stack Testing, from Sockets to Packets. In: *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 2013)*. 2560 Ninth Street, Suite 215, Berkeley, CA, 94710 USA: [b.n.], 2013, p. 213–218. Available at: https://www.usenix.org/conference/atc13/packetdrill-scriptable-network-stack-testing-sockets-packets.

[2] Corbet, J., Rubini, A. and Kroah-Hartman, G. *Linux device drivers*. 3rd ed. O'Reilly, 2005. ISBN 0-596-00590-3.

[3] Duke, M., Blanton, E., Eddy, W. and Braden, R. T. *A Roadmap for Transmission Control Protocol (TCP) Specification Documents* [RFC 4614]. RFC Editor, Sep 2006. DOI: 10.17487/RFC4614. Available at: https://www.rfc-editor.org/info/rfc4614.

[4] Fall, K. R. and Stevens, R. W. *TCP/IP Illustrated, Volume 1: The Protocols*. 2nd ed. Pearson Education, November 2011. Addison-Wesley Professional. ISBN 9780132808187.

[5] IEEE. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* [online]. [cit. 2022-04-22]. Available at: https://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10_06.

[6] Love, R. *Linux Kernel Development*. 3rd ed. Addison-Wesley, 2010. Developer's library: essential references for programming professionals. ISBN 9780672329463.

[7] Martin, J., Burbank, J., Kasch, W. and Mills, P. D. L. *Network Time Protocol Version 4: Protocol and Algorithms Specification* [RFC 5905]. RFC Editor, June 2010. DOI: 10.17487/RFC5905. Available at: https://www.rfc-editor.org/info/rfc5905.

[8] Melo, A. C. de. DCCP on Linux. In: Lockhart, J. W., ed. *Proceedings of the Linux Symposium* [online]. Ottawa, Ontario, Canada: [b.n.], July 2005, vol. 1, no. 1, p. 305–312 [cit. 2022-05-03]. Ottawa Linux Symposium. Available at: https://www.kernel.org/doc/mirror/ols2005v1.pdf.

[9] Mills, D. Internet time synchronization: the network time protocol. *IEEE transactions on communications*. New York, NY: IEEE. 1991, vol. 39, no. 10, p. 1482–1493. ISSN 0090-6778.

[10] Mills, D. *Computer Network Time Synchronization: The Network Time Protocol*. CRC Press, 2006. ISBN 9781420006155.

[11] Paxson, V., Allman, M., Chu, J. and Sargent, M. *Computing TCP's Retransmission Timer* [RFC 6298]. RFC Editor, June 2011. DOI: 10.17487/RFC6298. Available at: https://www.rfc-editor.org/info/rfc6298.

[12] Rosen, R. *Linux Kernel Networking: Implementation and Theory*. 1st ed. Apress, 2014. ISBN 978-1-4302-6196-4.

[13] Salzman, P. J., Burian, M., Pomerantz, O., Mottram, B. and Huang, J. *The Linux Kernel Module Programming Guide* [online]. 1st ed. April 2021 [cit. 2022-05-03]. Available at: https://github.com/sysprog21/lkmpg.

[14] Stevens, R. W. *Unix network programming. Volume 1: the sockets networking API*. 3rd ed. Boston: Addison-Wesley, 2004. ISBN 0-13-141155-1.

[15] Su, Z.-S. *Specification of the Internet Protocol (IP) timestamp option* [RFC 781]. RFC Editor, May 1981. DOI: 10.17487/RFC0781. Available at: https://www.rfc-editor.org/info/rfc781.

[16] Tanenbaum, A. S. and Wetherall, D. J. *Computer networks*. 5th ed. Pearson Prentice Hall, 2011. ISBN 0132126958.

[17] Watson, R. W. The Delta-t transport protocol: Features and experience useful for high performance networks. january 1989. Available at: https://www.osti.gov/biblio/5634768.

[18] Watson, R. W. Delta-t protocol specification: working draft. december 1981. DOI: 10.2172/5542785. Available at: https://www.osti.gov/biblio/5542785.

[19] Watson, R. W. Timer-based mechanisms in reliable transport protocol connection management. *Computer Networks (1976)*. 1981, vol. 5, no. 1, p. 47 – 56. DOI: https://doi.org/10.1016/0376-5075(81)90031-3. ISSN 0376-5075. Available at: http://www.sciencedirect.com/science/article/pii/0376507581900313.

[20] Welzl, M. *Network congestion control: managing internet traffic*. Chichester: John Wiley & Sons, Ltd, 2005. ISBN 9780470025284.

Appendix A

Contents of the included storage media

- `deltat` –
  - `src` – deltat kernel module implementation
  - `test` – simple Delta-t/TCP echo client–server application
  - `benchmark` – Delta-t/TCP file transfer application
- `wireshark` – wireshark LUA plugin adding Delta-t support
- `tcpdump` – modified version of `tcpdump` that can parse Delta-t segments
- `packetdrill` – modified version of `packetdrill`
- `doc` – latex source files of this thesis, including the `pdf` version of this work

Appendix B

Example packetdrill script

```
// Alleviate the timing constraints
--tolerance_usecs=500000
// Setup the enviroment for the test
`../defaults.sh`

// Open a socket and shrink its window on purpose
    0 socket(..., SOCK_STREAM, IPPROTO_DELTAT) = 3
   +0 setsockopt(3, SOL_SOCKET, SO_RCVBUF, [2048], 4) = 0
   +0 getsockopt(3, SOL_SOCKET, SO_RCVBUF, [4096], [4]) = 0
   +0 bind(3, ..., ...) = 0
   +0 listen(3, 1) = 0

// Establish a connection.
   +0 < deltat data D 0:1000(1000) win 32792
   +0 accept(3, ..., ...) = 4
   +0 > deltat ack - 1000:1000(0)
// The following segment should fill receiver's window
   +0 < deltat data D 1000:1460(460) win 32792
   +0 > deltat ack - 1460:1460(0) win 0     // Zero window adv.
// Trying to squeeze in another segment
   +0 < deltat data - 1460:2460(1000) win 32792
   +0 > deltat ack O 1460:1460(0)           // Not succesfull
   +0 < deltat rendezvous D 1460:1460(0) win 32792
   +0 > deltat ack - 1461:1461(0)
// Empty the receive buffer
   +0 read(4, ..., 1460) = 1460
   +0 > deltat ack R 66995:66995(0)         // Reliable-ack sent
// (Overflow) segment that dit not fit can be sent again
   +0 < deltat data D 66995:67995(1000) win 32792
   +0 > deltat ack - 67995:67995(0)
```

Figure B.1: Full version of a `packetdrill` script that simulates *rendezvous-at-sender* with window overrun.