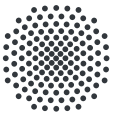


**University of Stuttgart**
Germany

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

# Design and Evaluation of System Concepts and Protocols for Lossless Hardware-Assisted Streaming of Real-Time Measurement Data over IP Networks

Leon Schürmann

| | |
|---|---|
| **Course of Study:** | Information Technology |
| **Examiner:** | Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel |
| **Supervisor:** | Dr. rer. nat. Frank Dürr |
| **Commenced:** | October 1st, 2021 |
| **Completed:** | April 20th, 2022 |

# Abstract

The problem of ensuring reliable delivery of data over unreliable transmission media is by no means unexplored. For instance, the Transmission Control Protocol (TCP) is a protocol designed to ensure reliable in-order delivery of data over an unreliable packet-oriented network service. However, TCP and similar protocols are most commonly implemented through software within operation systems designed to run on general-purpose compute hardware. For demanding measurement devices, however, implementations using custom logic implemented in application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs) can typically capture and process data at a greater rate and resolution compared to software systems.

This raises the question: how can measurement data be transferred from such a capture device to a remote system for storage and further processing, reliably and at a sufficient data rate? In an effort to answer the aforementioned question, this thesis analyzes preexisting mechanisms for reliable data transport over Ethernet and IP networks, as well as high-bandwidth measurement devices based on the example of a time-to-digital converter (TDC). Combining this knowledge, it presents HELIX, a network protocol and system architecture for reliable transmission of data, implemented through FPGAs. HELIX uses novel concepts and mechanisms to be efficiently implementable within FPGA-based systems, such as an integration of the transmitter's memory management architecture with the transport protocol itself.

# Kurzfassung

Die verlustfreie Übertragung von Daten über ein verlustbehaftetes Übertragungsmedium ist keineswegs unerforscht. Beispielsweise ist das Transmission Control Protocol (TCP) ein Protokoll, welches die zuverlässige Zustellung von Daten über einen verlustbehafteten und paketorientierten Netzwerkdienst garantiert, sowie die Reihenfolge übertragener Daten beibehält. Jedoch werden TCP und verwandte Protokolle herkömmlicherweise in Software unter Verwendung generischer Hardware implementiert. Um den Anforderungen anspruchsvoller Messanwendungen gerecht zu werden, ist für diese eine Implementierung durch spezialisierte Logikschaltungen vorzuziehen, typischerweise realisiert durch feldprogrammierbare Logikgatter (FPGAs) oder anwendungsspezifische integrierte Schaltungen (ASICs).

Dies wirft die folgende Frage auf: Wie können Messdaten, erhoben von solch einem Instrument, zuverlässig und ausreichend schnell an ein entferntes Computersystem gesendet werden, um diese dort weiterzuverarbeiten oder zu speichern? Um diese Frage zu beantworten, werden im Rahmen dieser Thesis existierende zuverlässige Transportprotokolle analysiert. Weiterhin werden solch FPGA-basierte Messinstrumente beispielhaft anhand eines sogenannten *Time-to-Digital Converter* (TDC) charakterisiert. Basierend darauf entwirft und präsentiert diese Thesis HELIX, ein Protokoll und eine Systemarchitektur zur zuverlässigen Datenübertragung, welche speziell auf eine effiziente Implementierung in FPGA-basierten Systemen ausgelegt ist. Dafür verwendet HELIX neuartige Mechanismen und Konzepte, beispielsweise die Integration und Auslagerung von Teilen der Speicherverwaltung am Sender in das Netzwerkprotokoll selbst.

# Contents

# Glossary

**bandwidth delay product** A network path's bandwidth delay product results from its available bandwidth, measured in bits per second, multiplied by the round trip time (RTT) of this path. As such, it is a measure to describe the amount of unacknowledged data a given network path will hold at any given time. Reliable transport protocols such as TCP have to transmit and buffer at least one bandwidth delay product worth of unacknowledged data to make effective use of a given network path's bandwidth [1]. 26

**bit error ratio** For a given transmission, the bit error ratio (BER) describes the ratio of erroneous bits to the total number of bits received. 9

**goodput** *Goodput* is a measure of network performance similar to throughput. However, whereas throughput describes overall bandwidth utilization, goodput only includes *useful* transmitted data. This means that generally, although depending on the context it is used in, goodput does not include overhead incurred through data retransmissions or protocol headers [2]. 51, 76

**round trip time** In networks, the RTT measures the time a given transmission takes to traverse the path from a source node to a destination node and the traversal time of a return transmission from the destination node to the source node. Thus it is a measure of bidirectional transmission delay. Depending on the network architecture, the path a transmission traverses from the source node to the destination node is not necessarily identical to the path from the destination node to the source node. 2, 13, 26, 49–52, 57, 68, 70, 71

**time jitter** Time jitter describes the dispersion of a signal in time or its digital representation, introduced through physical phenomena, time discriminators, and uncertainties introduced through measurement processes. An estimation of statistical time jitter is typically expressed as the root-mean-square deviation (RMSD) of deviations between sample timestamps from expected timestamps. 22

# Acronyms

**AIMD** Additive-Increase Multiplicative-Decrease 14

**ARP** Address Resolution Protocol 3

**ARQ** Automatic Repeat Request 12, 17, 28, 34, 38, 40–42, 47, 51, 53, 63, 64, 68, 72, 75, 77, 78

**ASCII** American Standard Code for Information Interchange 54

**ASIC** Application-Specific Integrated Circuit ii, iii, 1, 24, 27, 28

**AST** Abstract Syntax Tree 59

**BER** Bit Error Ratio v, 9

**CPU** Central Processing Unit 66

**CRC** Cyclic Redundancy Check 5, 8

**CSMA/CD** Carrier-Sense Multiple Access with Collision Detection 9

**DCCP** Datagram Congestion Control Protocol 17, 18

**DDR3-SDRAM** Double Data Rate 3 Synchronous Dynamic Random-Access Memory 58, 59, 76

**DRAM** Dynamic Random Access Memory x, 26, 27, 30, 46, 58–61, 63–66, 74

**DSL** Domain-Specific Language 59

**FEC** Forward Error Correction 78

**FIFO** First-In First-Out ix, 3, 16, 49, 53, 61, 66–68, 73, 74

**FPGA** Field-Programmable Gate Array ii, iii, 1–3, 22–28, 30, 33, 34, 47, 48, 50, 54, 56, 58, 59, 61, 66, 70, 74, 77, 78

**HDL** Hardware Description Language 1, 59, 73

**HELIX** Hardware-Enabled Lossless Internet Information eXchange ii, iii, ix, 2, 29, 50, 53–57, 59, 68–70, 72, 74–78

**HTTP** HyperText Transfer Protocol 16, 19

**ID** Identification Number 41–43, 45, 46, 51–56, 62–65, 70, 74, 75

**IEEE** Institute of Electrical and Electronics Engineers 1, 5, 50, 59, 70

**IP** Internet Protocol ii, 1–3, 5–10, 12, 16, 18, 19, 25, 27, 28, 31, 42, 44, 49, 53, 56, 59, 77

**JCS** JavaScript Object Notation Canonicalization Scheme 54, 55

**JSON** JavaScript Object Notation 55

**LIDAR** Light Detection and Ranging 21

**MAC** Media Access Control 3, 25, 50, 59

**MII** Media Independent Interface 25

**MTU** Maximum Transmission Unit 9

**NIC** Network Interface Card 7, 16, 25, 50, 70

**PHY** Physical Layer 3, 25, 59

**PTP** Precision Time Protocol 50, 70

**RAM** Random Access Memory 26

**RFC** Request for Comments 5, 6, 10, 12, 13, 15–17, 19, 30, 36, 46, 49, 52, 54, 78

**RMSD** Root-Mean-Square Deviation v

**RTO** Retransmission Timeout 57

**RTT** Round Trip Time v, ix, 2, 13, 26, 49–52, 57, 68, 70, 71

**SCTP** Stream Control Transmission Protocol 19, 20

**SFP+** Enhanced Small Form-Factor Pluggable 3, 58

**SHA-256** Secure Hash Algorithm 2 with 256 Bit Output 54, 55

**SO-DIMM** Small Outline Dual Inline Memory 58

**SoC** System on a Chip 58

**SRAM** Static Random Access Memory 26, 62

**TCP** Transmission Control Protocol ii, iii, v, ix, 2, 3, 10–20, 26, 28–33, 36, 41, 42, 47–50, 52–54, 56, 77–79

**TDC** Time-To-Digital Converter ii, iii, 1, 21–24, 48

**TIMELY** Transport Informed By MEasurement of LatencY 49–51, 68, 70

**UART** Universal Asynchronous Receiver-Transmitter 59

**UDP** User Datagram Protocol 17, 18, 53, 54, 59

**XGMII** 10 Gigabit Media Independent Interface 3, 25, 59

# List of Figures

# List of Tables

# 1. Introduction

With advances in science and technology come the requirements to measure physical phenomena and acquire measurement data at an ever-increasing rate and precision. This trend is observable in a variety of settings, such as industrial applications and research projects, from those in computer vision to quantum computing to particle physics. However, for any of these applications to work, it is insufficient for measurement and data capture devices to only acquire such data; they must further be transmitted to a remote system and processed accordingly.

An example of such a measurement device is a time-to-digital converter (TDC). It converts rising and falling edges in an incoming digital electrical signal into a series of timestamps. Depending on the application area, such devices must feature a high timestamping accuracy as well as a high sustained and burst sample rate. However, these properties directly translate to an increased output data rate of the device. Furthermore, when observing given phenomena through a measurement series, it often is crucial not to lose any captured data points in order to be able to draw valid conclusions about the observed phenomena.

Various networking technologies and protocols exist to transfer captured information from such a device to a host for storing and further processing. However, the combination of the Internet Protocol (IP) as a protocol for a packet-oriented data transfer between networked endpoints, together with Ethernet-based links, proves to be a particularly attractive option: on the one hand, the interoperability of IP overcomes boundaries between underlying network layers, different device vendors, operating systems, and system architectures [3]. On the other hand, the Institute of Electrical and Electronics Engineers (IEEE)'s 802.3 Ethernet is a widespread network layer protocol, defining a collection of high-speed and high-distance capable physical transport layers, with specifications reaching 400 Gbit/s over at least 500 m [4, sec. 8, 124]. The availability of affordable high-speed Ethernet- and IP-capable hardware, along with IP's flexibility and interoperability, make the combination of Ethernet and IP a popular and widespread data exchange mechanism. Furthermore, Ethernet and IP are also prevalent within integrated hardware systems such as application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs), with preexisting and open-source hardware description language (HDL) *cores* implementing required protocol processing logic, as well as FPGA transceiver hardware primitives providing certain Ethernet physical layer implementations. Because measurement devices such as TDCs are fundamentally hardware systems implemented through ASICs or FPGAs, it appears viable to transfer captured measurement data via Ethernet and IP, without intermediate general-purpose compute systems.

Nonetheless, IP and Ethernet cannot alone meet all requirements imposed by such measurement devices. Most importantly, IP does not guarantee data integrity or prevent data reordering or erasure [3, p. 3]. For applications where in-order, reliable, and unmodified arrival of transferred data is crucial, additional protocols on top of IP must establish these characteristics. Perhaps the most prevalent such protocol to guarantee reliable data transmission over IP is the Transmission Control Protocol (TCP) [5; 6, p. 263]. Although TCP works well for many applications, it has a multitude of issues when mapping it to hardware- or FPGA-based implementations and when faced with the requirements of measurement systems such as the ones described above: for instance, it is fundamentally subject to so-called *head-of-line blocking* through data structures it enforces at the transmitter. TCP also introduces significant complexity through its connection management architecture and extensible nature. A basic presumption of this thesis is that a protocol specifically designed to be implemented in hardware or FPGA-based systems, and featuring characteristics targeted towards such measurement systems, will differ from established protocols designed primarily to be efficiently representable through software implementations. Specifically, such a protocol can employ a simpler connection management architecture utilizing existing low-performance reliable communication channels to establish connections, reduce inherent extensibility and variability in the protocol and its implementation, and employ an efficient memory management architecture tightly integrated with the protocol.

To that end, this thesis presents a novel system architecture and network protocol named HELIX (Hardware-Enabled Lossless Internet Information eXchange). Similar to established technologies such as TCP, it uses an IP transport channel. However, HELIX explicitly targets requirements identified for hardware- or FPGA-based measurement and data-capture devices to effectively utilize the underlying data link layer while ensuring data arrival and integrity, in addition to posing minimal overhead and complexity onto both the capture device and the receiving host system. In that, HELIX introduces a novel buffer and memory management architecture specifically designed to work with FPGA systems. This architecture is tightly integrated with the network protocol, cooperating to offload parts of the memory management overhead to the network and receiving host. Furthermore, HELIX benefits from the integration of innovations of TCP, such as round trip time (RTT) based congestion control mechanisms.

In particular, the contributions of this thesis include:

**Memory Model for In-transit and Retransmit Data** A hardware-optimized model for organizing data transferred over a network connection. It can distinguish between data in transit and data known to be lost during transmission. It efficiently implements all operations required to be usable as a buffer mechanism for a reliable transport protocol, while imposing certain restrictions on this protocol. It differs from existing memory models by cooperating with the receiver to optimize certain operations, specifically in transferring memory addresses along with data, which are sent back by the receiver to enable efficient processing of selective acknowledgments.

**LiteEth 64 bit Data Path** Support for a 64 bit data path in the LiteEth FPGA Ethernet media access control (MAC) core to enable processing 10 Gbit/s Ethernet data at clock frequencies achievable in FPGAs today. This work has been based on preexisting efforts from open source contributions and was implemented in cooperation with David Sawatzke of the University of Stuttgart. Specific contributions include adapting the packet header parsing (*Depacketizer*) and packet header generation (*Packetizer*) logic to work with variable-width data paths, in addition to other required changes to the Address Resolution Protocol (ARP) and *Wishbone Bus* integrations.

**LiteEth XGMII Interface Support** A 10 Gigabit Media Independent Interface (XGMII) frontend for the LiteEth FPGA Ethernet MAC core, including an implementation of the *deficit idle count* mechanism (compare [4, sec. 4, 46.3.1.4]). This is required to interface with 10 Gbit/s Ethernet physical layer (PHY) transceiver devices responsible for implementing the Ethernet links' electrical or optical interfaces and encoding schemes.

**FIFO-based Memory Address Hazard Detection Scheme** An algorithm and implementation to detect hazard conditions in pending memory write operations for pipelined memory access controllers, based on pending write addresses stored in a first-in first-out (FIFO) buffer. The developed algorithm has a constant time and space complexity. In particular, this scheme is useful to detect potentially hazardous addresses when using multiple ports of the LiteDRAM memory controller.

**NetFPGA-SUME Board Definition for LiteX** Support of the NetFPGA-SUME FPGA development board, along with peripherals such as the 10 Gbit/s Ethernet Enhanced Small Form-Factor Pluggable (SFP+) ports, within the LiteX SoC generator framework.

The remainder of this thesis is structured as follows:

First, **Chapter 2** establishes some important background knowledge of reliable communication systems, as well as violations of these aspects within IP networks through phenomena such as packet reordering and packet loss. The chapter continues by giving a detailed introduction to the mechanisms and concepts established within TCP and related protocols.

Following this, **Chapter 3** proceeds to describe the system model, including the target application area, provided interfaces, and inherent restrictions of these devices.

Based on the background knowledge and system model, **Chapter 4** illustrates the problem at hand.

In an effort to provide solutions for the problem identified, **Chapter 5** explores the design space of viable solutions and presents the developed novel memory model architecture, as well as other important concepts.

**Chapter 6** implements the design elaborated in the previous chapter while documenting noteworthy challenges during the implementation.

Based on this implementation, **Chapter 7** validates that the developed solution is correct and applicable to the problem at end. It further presents a benchmark outlining the solution's performance.

Finally, **Chapter 8** concludes this thesis and discusses potential future work.

# 2. Background and Related Work

The problem of establishing a reliable data transmission channel based on an unreliable one is by no means unexplored. Works such as Shannon's "A Mathematical Theory of Communication" have established the space of information theory, and with that the foundations of reliable communication of digital information [7]. Based on this and other works, different models and algorithms for data encoding, recovery and reliable transmission of data in general have been established. Because of its ubiquitous nature, IP is a particularly popular example of a potentially unreliable transmission protocol to serve as a basis to build mechanisms and protocols for reliable data transmission. This chapter shall establish a basis to understand the reliability constraints imposed by this thesis' target application areas and compare preexisting technologies in this space.

## 2.1. Communication Channel Reliability

When transmitting data over any given channel, there is a statistical chance for this transmission to be corrupted. Such corruption may happen because of physical (electrical or optical) noise and interferences with the transmission itself, resulting in a degradation of the transferred signal over the transmission line, issues with the transmission channel itself, or through intermediate stations on the end-to-end data path. While mechanisms exist to reduce the probability of uncorrectable data corruption occurring on a given transmission channel, in the form of adding controlled redundancy in the transmitted data, fundamentally, any data transmission cannot be guaranteed to traverse the path between sender and receiver without experiencing data corruption [8, p. 354]. This fundamental truth is acknowledged by standards and protocols such as IEEE 802.3 Ethernet and IP. For instance, Ethernet requires transmitted frames to be followed by a so-called *Frame Check Sequence* in the form of a cyclic redundancy check (CRC) sequence, a mathematical error detection mechanism [4, Sec 1, 3.2.9; 8, p. 453]. Request for Comments (RFC) 791, describing the Internet Protocol, states that it "[...] does not provide a reliable communication facility" [3].

While it is trivial to establish that the correct recovery of data transmission over a noisy channel is inherently a stochastic process [7, pp. 406–407], a given protocol's *reliability* guarantees over such a channel may describe a broad set of different characteristics. In order to analyze existing protocols concerning their guarantees, *reliability* in the context of communication must be defined further. Spinelli defines reliable data communication as "[...] the delivery of some set of information packets from a data source to a data sink, in order, and without any lost, inserted, or duplicated packets" [9, p. 10]. This definition fails

to explicitly state that an additional requirement of reliable data communication may be the unaltered (intact) delivery of information packets. Based on this definition, the following basic characteristics of a reliable communication method can be established:

- **Lossless delivery:** Every unit of data sent by the transmitter must eventually be received by the receiver. There must not be any *loss* or *erasure* of information.
- **No insertion:** The receiver of a transmission must only receive data which was indeed sent by the transmitter.
- **No duplication:** The receiver of a transmission must, for every datum transmitted, only ever receive a single copy of this datum.
- **In-order delivery:** All units of data sent by the transmitter must arrive at the receiver in exactly the order in which they were sent.
- **Intact / unaltered delivery:** All units of data sent by the transmitter must arrive at the receiver exactly how they were sent. That is, the data received must represent the same information as the transmitted data.

Fundamentally, a reliable communication channel may be synthesized atop an unreliable transmission channel, as analyzed by Spinelli [9]. This implies that the design decisions and the complexity of a protocol augmenting these requirements depend on the inherent properties of the underlying communication channel, and the desired properties of the resulting communication channel.

## 2.2. Reliability of IP Networks

IP (RFC 791) does not guarantee any of the above characteristics to be upheld. While some characteristics such as *no insertion* and *intact / unaltered delivery* are mostly dependent on the underlying transport protocol(s), others are in direct relation to IP routing protocols and thus by extension specific to IP. For instance, essentially any link layer transmitting data through electrical, optical or electromagnetic signals is subject to noise and interference over the transmission channel. Ultimately, this noise or interference may degrade the signal substantially, up to a point where the receiver either cannot decode the channel-coded signal $c_{m,\text{recv}}$ towards a valid codeword $c_m \in M = \{(c_{m_0}, \ldots, c_{m_n}) \mid c_{m_i} \in \{0,1\}\}$ any longer (*loss / erasure*, if received codeword $c_{m,\text{recv}}$ is not from the set of valid codewords $M$), or may even decode it towards a different, valid codeword (violating *intact and unaltered delivery*, when $c_{m,\text{recv}} \in M$, but $c_{m,\text{recv}} \neq c_{m,\text{sent}}$). However, primitive physical channels are generally less likely to change the order of arrival of information, or to duplicate transmitted information; although such phenomena might be physically possible through pathologic characteristics such as signal reflections [8, p. 9]. Because different reliability characterists depend on different aspects of both the physical transmission media and intermediate nodes in between primitive physical channels, the behavior of larger networks consisting not only of a single physical channel, but multiple participating network nodes and edges, is of interest. Even more so, for a given reliable transport protocol it

does not always make sense and it might not always be possible to distinguish individual violations of reliability when using IP [10]. Instead such a protocol may view IP as an opaque mechanism to transfer data from one application to another. Hence, reliability characteristics of IP shall be considered in an end-to-end fashion, from a sending to a receiving application [10].

With the Internet Protocol being this thesis' explicit target protocol, and based on the communication channel reliability characteristics analyzed in the previous section, this section shall further focus on the fundamental causes and effects of two major violations of reliable communication in relation to IP: packet reordering and packet loss.

### 2.2.1. Packet Reordering in IP Networks

In particular, occurences of the symptom of packet reordering in IP networks are inherently dependent of the entire network architecture in question, that is the participating nodes and paths in between these nodes. Packet reordering, and in particular its consequences for higher-level protocols aiming to eliminate this characteristic, has been subject of extensive research [11; 12; 13, pp. 109–148; 14]. Of significant discussion is the question whether occurences of packet reordering must be seen as an inherit property of IP networks, or whether packet reordering should rather be considered a pathological phenomenon; and by extension the question whether it must be handled also within smaller, controlled networks. A basic and valid assumption of these analyses is that reordering of data within a given IP packet does not have to be considered. This is because IP divides information into discrete collections of data represented through a byte sequence, so called *packets*. If the order of encoded information (bits) changes within such a packet, this is essentially indistinguishable from—and thus for our intents equivalent to—data corruption. In contrast, when only the order of arrival of otherwise unmodified IP packets changes, all information is still delivered intact. When using Ethernet, reordering of data within an IP packet is prevent by the employed *Frame Check Sequence* [4, Sec 1, 3.2.9].

According to Bennett et al., packet reordering is caused by intermediate network elements on an end-to-end path through parallelism in network elements. For instance, when a single logical path is comprised of multiple physical paths, each of these physical paths may be subject to different queues in the active network elements and thus packets may experience varying delays through these different paths. When this form of parallelism leads to packets of the same *flow* to take different physical paths, with a flow defined as the quintuple of source IP, destination IP, transport protocol, source port, and destination port, this may change order of arrival of data within this flow [11]. In addition to parallelism in the links or during processing within intermediate network devices, factors such as network faults, improper configuration and faulty software may contribute to packet reordering in the network [15]. This symptom is not necessarily limited to intermediate network elements: the receiver's system and network interface card (NIC) architecture may also introduce packet reordering as an undesirable consequence of parallelism in packet processing [14].

Common among analyses of packet reordering is that they generally strive to determine how packet reordering affects the Internet at large, through measuring over long, multi-hop paths [12] or with Internet backbone network equipment [11]. While usage of IP as a network protocol theoretically allows data to travel over arbitrary paths through the Internet, it is much more likely for measurement data to traverse a limited set of controlled paths between the measurement device and receiving host. It is expected that a reduced complexity in the network, along with hardware performing parallel packet forwarding and processing on a flow-granularity, causes packet reordering to be less prevalent compared to the overall Internet. Nonetheless, given IP does not make any strict guarantees concerning packet reordering, a protocol providng a reliable communication channel atop of IP networks must be tolerant of it. In addition to IP packet reordering as a violation of reliability guarantees, if a transport protocol-issued retransmission of lost packets is identical to the original transmission of these packets, the reception of such a retransmission can be indistiguishable from unintential packet reordering. However, it must be assumed that in this case the transport protocol features sufficient mechanisms to handle such out-of-order arrival of packets; thus these mechanisms shall be described along with the protocol design, if necessary.

### 2.2.2. Packet Loss in IP Networks

A different but common phenomenon observed through IP networks is packet loss. It can be a symptom of many effects caused by the sender, receiver, or transmission channel. Nonetheless, there are two prevalent classes of issues in IP networks that manifest themselves in packets sent by the sender not arriving at the receiver.

As outlined previously, data transmissions using the IP protocol are divided into atomic units called packets. Mandatory for all IP packets is to start with an IP header, which has a different layout and inherent properties depending on the version of the IP protocol in use. Of importance for the phenomenon of packet loss is the inclusion of a checksum for header data within the fourth version of the IP protocol [3, p. 14]. Furthermore, underlying transport protocols such as Ethernet may mandate additional checksums over either partial or entire transmission units. For instance, Ethernet mandates a 32 bit *Frame Check Sequence* (32 bit CRC) be appended to every Ethernet frame [4, Sec. 1, 3.2.9]. Such integrated integrity mechanisms allow detecting but not necessarily correcting a limited number of errors in transmission caused by noise or interference introduced by the transmitter, receiver, or channel. When such an error is detected and cannot be reliably corrected, the receiver or intermediate network stations are obliged to discard the network packet [3, p. 3; 4, Sec. 1, 4.1.2.1.2]. Considering the end-to-end transmission path, this manifests itself as packet loss.

Packet loss can occur even without data errors introduced by the sender, an intermediate system, the receiver, or the transmission channel. Fundamentally, the design of IP enables and encourages multi-hop communication and usage of shared transmission channels for multiple, independent data flows. Thus intermediate network stations can receive, inspect, transmute, and finally transmit IP packets to ultimately reach their final destination. This

process is known as *packet switching*; for IP it is referred to as *routing* [16, p. 4]. Because any physical transmission channel has a limited capacity, when multiple flows' packets are destined to be transmitted over a single physical channel, intermediate network stations may need to hold on to packets whose onward transmission channel is currently occupied [17, p. 356]. However, any intermediate network station can only enqueue a limited amount of data and number of packets at any given time. Thus, reaching these limits will ultimately lead to intermediate network elements discarding packets. This symptom is known as network congestion, also manifesting as packet loss.

While both described phenomena appear in practice, it is also essential to consider the likelihood of occurrence and the relation of packet loss to the overall network and link utilization. For instance, 1000BASE-T (1 Gbit/s) Ethernet mandates maintaining a bit error ratio (BER) of less than or equal to $10^{-10}$ [4, Sec. 3, 40.1.1]. Assuming a maximum transmission unit (MTU) of 1500 byte, a full-size Ethernet frame of 1518 byte therefore has a likelihood of experiencing corruption in any of its data bits of less than $1 \times 10^{-10} \frac{\text{error}}{\text{bit}} \cdot 1518 \frac{\text{byte}}{\text{frame}} \cdot 8 \frac{\text{bit}}{\text{byte}} \approx 1.2144 \times 10^{-6} \frac{\text{error}}{\text{frame}}$, meaning at most one in every 823451 Ethernet frames is corrupted on average, assuming an intact and compliant link. On the one hand, transmission channel-induced data corruption on a dedicated transmission medium is typically a random process independent of network or link utilization. On the other hand, depending on the precise carrier in use, link arbitration schemes such as carrier-sense multiple access with collision detection (CSMA/CD) can establish a direct relation of network utilization by other participants to the probablility of experiencing transmission collisions and thus data corruption [4, Sec. 1, 4.1.2.2]. However, for the purposes of this thesis, all bidirectional links between network elements are assumed to be dedicated channels, with individual collision domains for each directional link, and having negligible susceptance to interferences caused by other transmissions and noise sources. End-to-end packet loss caused by network congestion can be more reliably correlated with network and link utilization [18]. However, intermediate network elements have evolved to employ complex queueing models, which makes direct attribution of losses to exceeding link capacities difficult or impossible.

In practice, on dedicated electrical and optical point-to-point links, packet loss due to corruption can be assumed to be rare, compared to loss events caused by network congestion [18]. As this thesis does not target wireless links or bus topologies, where this assumption is shown to be problematic [19], it appears viable to focus on network congestion being the primary contributor to packet loss.

## 2.3. The Transmission Control Protocol

Section 2.1 and Section 2.2 explore the characteristics and theory of reliable communication, and how symptoms such as packet loss and packet reordering relate to IP. Based on that knowledge, this and subsequent sections analyze preexisting protocols implementing a reliable communication channel over the unreliable IP network service, by supplementing all or a limited subset of the characteristics of reliable communication.

The Transmission Control Protocol (TCP) is a protocol designed to establish a reliable communication service between applications in a multinetworked environment, such as provided by IP [20]. Specified through RFC 793 in 1981, it has developed to become the prevalent protocol for establishing reliable communication channels over IP-based networks. Fundamentally, TCP is concerned with imposing connection management and reliability mechanisms onto a packet-oriented datagram protocol, in order to compose a reliable connection-oriented communication channel with a byte-stream interface. Even though the basic ideas and principles of TCP can be viable for a wide range of underlying protocols, it is designed with the assumption of using IP as its underlying protocol layer [20, p. 3]. Through the mechanisms described hereafter, TCP aims to establish the reliable communication characteristics of *lossless*, *intact*, *in-order* delivery without *duplication* of transferred data [20, p. 4]. Although not explicitly specified, by nature of its design TCP provides basic resilience to *insertion* caused by underlying channel artifacts, but these protections are insufficient to prevent insertion of arbitrary data through a malicious actor. Since the initial specification through RFC 793, TCP has been modified and extended through various other RFCs and scientific publications, of which not all have seen widespread adoption. This thesis considers modern TCP implementations, that is, ones compliant with the basic protocol specification and supporting common extensions. The remainder of this section will explore different aspects of TCP, in particular its mechanisms to compose a reliable communication facility over the unreliable IP transport.

### 2.3.1. TCP Connections

In order to provide the concept of a *connection* atop of a fundamentally *packet-switched* architecture as imposed by IP, TCP introduces a connection management architecture [20, pp. 5, 30–39; 17, pp. 560–565]. A TCP connection is a bidirectional data channel for which the protocol ensures that the desired aspects of reliable communication are upheld. TCP endpoints are referred to as *sockets*, identified by their host IP address and a (unique per IP) 16 bit numerical service identifier called *port*. Each connection is established between two applications running on network hosts, hence connecting two sockets. A single socket on one host may be connected to multiple sockets on remote hosts, however TCP solely supports unicast messaging, meaning that each connection involves exactly two sockets, one socket (source) cannot transmit data to multiple sockets (destinations) in a single operation [20, p. 5; 17, p. 553]. TCP introduces sophisticated mechanisms for connection establishment; a detailed figure outlining the various connection states and possible transitions defined by TCP can be found in Appendix A. Typically, a TCP connection is established through a *passive open* or *listen* request, opening a local socket with an unspecified foreign socket. The host providing this passive open or listening socket is referred to as a *server*. An application wishing to establish a connection with this listening socket, called *client*, issues an *active open* request and specifies the foreign socket address. The server is informed of this connection request and, if accepted, a connection is established [20, pp. 11–12, 23; 17, pp. 562–564]. An established connection can be used for bidirectional (*full duplex*) data transfer between the two sockets [20, p. 10]. Each socket can request for the connection to be closed, of which it will inform the respective remote socket. The connection concept introduced by TCP provides an envelope for the transported byte

streams in either direction and is used by implementations of TCP to manage connection state and provide the aforementioned guarantees for transferred payload data. This means that all guarantees with respect to reliable communication established by TCP are fundamentally limited to the channel established through a single connection. TCP does not, for instance, ensure *in-order* delivery of data between two hosts when transferred through different connections. While TCP is a bidirectional, full-duplex capable protocol, with a connection fundamentally comprising two directional data channels, in accordance with the thesis target application area the remainder of this section will focus on TCP used as a primarily unidirectional communication mechanism.

To transmit data through a connection between sockets, TCP takes incoming bytes to transmit (with a byte defined as an 8 bit *octet*) and splits the byte sequence, dividing it into chunks of data [20, p. 4]. A TCP header containing metadata and control signals is prepended to each formed chunk in order to compose TCP *segments*. The precise offsets of chunk boundaries within the incoming byte sequence and how much outstanding untransmitted data a socket can accept (buffer) is at the discretion of the TCP implementation at hand and not necessarily exposed to the sending application [20, p. 4]. Internally, TCP uses the synthesized segments as an encapsulation mechanism to translate the provided stream-oriented interface onto a fundamentally packet-oriented interface.

To ensure that data is reliably transferred from sender to receiver, each segment is labeled with a *sequence number*, indicating the offset of the first byte within the outgoing data stream transferred through this segment [20, p. 10]. Segments contain further control and status messages, such as an acknowledgment number. To account for violations of reliability in the underlying network layer, such as packet loss, TCP can recover from loss events by retransmitting data. Acknowledgments are used by the receiver of a TCP stream to indicate that all data up to but excluding the byte offset indicated through the acknowledgment number have been received [20, p. 10]; thus TCP's acknowledgments are *positive acknowledgments*. A TCP sender can use information received through such acknowledgments to retransmit data of which it does not know whether they have been received by the receiver. TCP segments also serve as a data unit for two mechanisms of major importance: *flow control* and *congestion control* [20, p. 4].

### 2.3.2. TCP Flow Control

Flow control is a mechanism to limit the amount of data a sender may transmit towards the receiver, to avoid a fast sender from overwhelming a slow receiver: if the receiver application cannot process the sustained incoming data rate in time, this will cause the implementation's receive buffers to be incapable of holding further data; requiring a mechanism to signal the transmitter that new data cannot be accepted. TCP flow control provides a solution to this issue by introducing the concept of a receive window, illustrated in Figure 2.1. The receiving end of a TCP connection announces a receive window size to the transmitter as part of an acknowledgment message, represented through the number of bytes after the acknowledge sequence number which the receiver is prepared to accept [20, pp. 15–16, 42]. The receiver is free to announce a different window size with each acknowl-

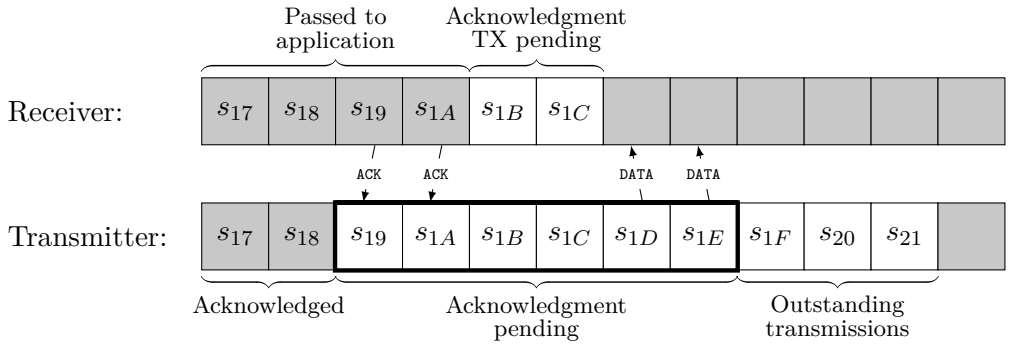



Figure 2.1.: TCP *receive sliding window* illustrated through a sequence of segments transferred from transmitter to receiver, with a maximum window size of 6 segments.

edgment [17, pp. 565–566]. The transmitter is obligated to keep track of the transmitted segments, or at least their sequence number, and must ensure that all transmitted data beyond the offset indicated through the last received acknowledgment number must not exceed the announced receive window size. Because this window is specified as a relative offset of the acknowledge sequence number, it is shifted to include further data if data at the start of the previous window is acknowledged. Therefore, this mechanism is referred to as a *sliding window*.

### 2.3.3. TCP Congestion Control

Whereas flow control is a mechanism for the receiver to limit the amount of data it needs to buffer and to control the transmission data rate, congestion control refers to a collection of mechanisms to avoid exceeding available network resources. As analyzed in Section 2.2.2, packet loss in IP networks is commonly caused by exceeding the available bandwidth of a given link among a path of network links. The set of problematic or limiting links on a given path are referred to as the *bottleneck links*. If a TCP sender were to send incoming data as fast as possible, only accounting for the receive window mandated as part of the flow control mechanism, it is at risk of loosing a significant number of packets when it exceeds the bottleneck link's bandwidth limitations. While TCP provides automatic repeat request (ARQ) mechanisms to schedule retransmissions of (assumed to be) lost segments, a significant number of lost packets drastically reduces TCP's overall efficiency. Congestion control mechanisms aim to control the bandwidth utilized by a TCP connection, in an effort to minimize loss events while fully utilizing a *fair* share of the available end-to-end bandwidth [18]. TCP has been designed with fairness as a goal, such that multiple TCP connections can share links and retrieve an approximately equal share of bandwidth over time. In contrast to the fundamental flow control mechanism illustrated above, congestion control mechanisms were not introduced with the original specification of TCP within RFC 793, but have been introduced as part of the `4BSD` TCP implementation and described by Jacobson in [18]. In the meantime, they have been proposed as RFC standards 2001, 2581 and the draft standard RFC 5681.

In particular, these documents describe four interworking congestion control mechanisms employed by modern and efficient TCP implementations: *slow start*, *congestion avoidance*, *fast retransmit* and *fast recovery*. These mechanisms shall be explained briefly:

**Slow start** describes an algorithm to gradually increase TCP's utilized transmission bandwidth. If the transmitter were to fully utilize its available link bandwidth at the start of the connection or on recovery from a loss event, this would likely result in a significant number of loss events at the bottleneck links. As a consequence, given the transmitter will only receive acknowledgments for consecutive segments arrived at the receiver, the receive sliding window will only partially move forward and requires retransmissions of lost segments. This behavior will repeat for newly transmitted segments, if the transmission rate is not throttled.

Jacobson proposes *slow start* as a solution to this issue [18]. It introduces a *congestion window*, limiting the amount of unacknowledged data in the network at any given time. Where the receive window's purpose is to avoid overwhelming the receiver, the congestion window is used to overwhelm the network, and specificically the bottleneck link capacity. In contrast to the receive window, the size of this window is controlled by the sender. To implement slow start, the size of this congestion window is increased by one segment on arrival of an acknowledgment for new data. The transmitter uses the minimum of the congestion and receive windows as a bound on in-transit non-acknowledged data [21]. This algorithm causes TCP to converge towards the path bandwidth by avoiding excessive loss events preventing convergence.

**Congestion avoidance** is used in place of the slow start algorithm once the size of the congestion window has reached a certain threshold [21]. Whereas TCP slow start is a mechanism designed to establish reliable data flow over a path with unknown characteristics, congestion avoidance aims to retain data flow on a link and converge towards utilizing the maximum available path bandwidth [18].

During congestion avoidance, multiple algorithms exist to increase the congestion window size. In general, as per RFC 5681, these must not increase the congestion window size by more than the maximum segment size once per round trip time (RTT). However, upon detecting a segment loss as determined by the TCP retransmission timer, the congestion window size threshold of when to use congestion avoidance is adjusted to either half of the amount of unacknowledged data in the network or twice the maximum segment size, whichever is larger. Also, TCP will reset the congestion window size to one full-sized segment and thus resort back to the slow start mechanism [21].

Combined, slow start and congestion avoidance cause a TCP connection to probe the network path's available bandwidth continuously. With slow start responsible for exponentially increasing the sender's congestion window size up to a certain threshold and congestion avoidance linearly increasing the congestion window size until loss events occur, TCP manages to achieve sufficient link utilization to approach avail-

able network resources quickly. At the same time, it avoids exceeding the available network resources which would cause significant loss events. By having TCP congestion avoidance influence the threshold of switching exponential congestion window growth to linear growth, it can efficiently recover link utilization towards a known-safe level in response to loss events in logarithmic time. Given that TCP effectively implements linear growth of the congestion window during regular operation and reduces the congestion window size by approximately half of its value in case of loss events, this mechanism is known as an additive-increase multiplicative-decrease (AIMD) algorithm [22]. A trace of the congestion window size over time resembles a sawtooth pattern, probing the available path bandwidth (compare Figure 2.2).

**Fast retransmit and fast recovery** are cooperative mechanisms between a TCP sender and receiver to handle certain transmission anomalies more efficiently than they would be handled using only congestion avoidance and slow start algorithms at the sender. Specifically, they are designed to efficiently handle network behavior which manifests as an out-of-order arrival of segments at the TCP receiver. This may either be a true reordering of data or a loss of an intermediate segment.

Upon receiving an out of order segment, a TCP receiver is obliged under fast retransmit to send a duplicate acknowledgment of the last consecutive segment for each such out of order segment. If the sender receives three identical duplicate acknowledgments, it can assume that at least the subsequent segment has been lost and should schedule it for immediate retransmission. However, an out-of-order arrival of a truly reordered packets will—following these rules—also issue a duplicate acknowledgment, even though it is not truly lost. Thus, upon receiving a segment aligned to the last consecutive segment received and filling in a gap in the sequence space, the receiver should send an immediate acknowledgment. This allows the sender to, in many cases, avoid an unecessary retransmission of reordered data [21].

Because a retransmission in response to a loss event announced by the receiver using duplicate acknowledgments fundamentally indicates that packets *after* the lost packets have been successfully received, reverting back to the slow start algorithm hinders effective use of available bandwidth. Instead, *fast recovery* is used to govern transmission of new data until receiving a non-duplicate acknowledgment. During this phase, TCP sets the congestion window size to the threshold as implemented in congestion avoidance. However, it furthermore uses the number of duplicate acknowledgments received as an indication of data leaving the network, and thus increments the congestion window size for each duplication acknowledgment received accordingly [21].

These four mechanisms are essential to TCP being able to effectively utilize available path bandwidths, while also fairly sharing available bandwidth between different TCP connections. It has been shown that an additive-increase multiplicative-decrease (AIMD) scheme converges towards an approximate equal and thus fair utilization of path bandwidth, while schemes implementing multiplicative-increase additive-decrease, additive-

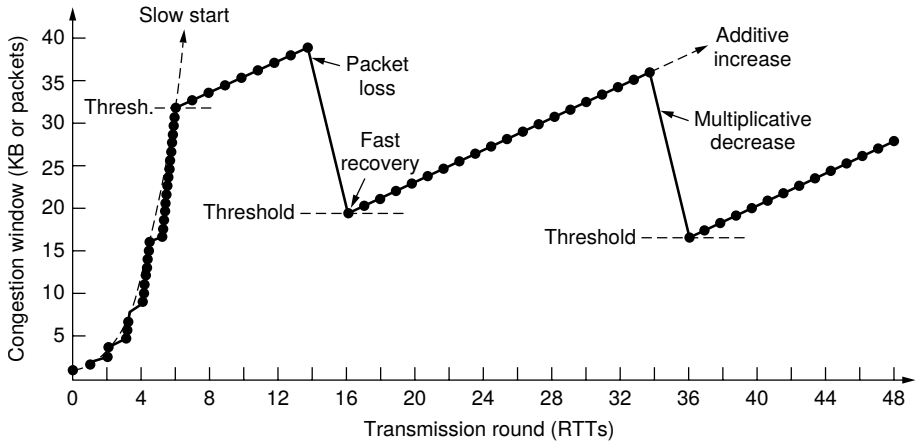



Figure 2.2.: Illustration of the TCP congestion window size over time, outlining usage of the slow start, congestion avoidance, and fast recovery mechanisms. Without fast recovery, the packet loss at `RTT` = 14 would cause a fallback to slow start again, with an adjusted threshold. Figure taken from *Computer Networks* by Tanenbaum and Wetherall [17, p. 579].

increase additive-decrease, or multiplicative-increase multiplicative-decrease will not [22; 17, pp. 537–539].

### 2.3.4. TCP Selective Acknowledgments

An optional extension to the basic Transmission Control Protocol as specified in RFC 793, RFC 2018 is a proposed standard introducing *selective acknowledgments* for TCP [23]. Selective acknowledgments improve over TCP's acknowledgments, which through their cumulative nature are necessarily consecutive to the last in-order received segment. While regular TCP cannot acknowledge received segments which are not consecutive to the left edge of the receive window, selective acknowledgments allow the receiver to inform the sender about precisely which segments have been received, such that only lost segments must be retransmitted. This is done by acknowledging data in so-called *blocks*, described by the TCP sequence number of the *left edge* (chronologically oldest byte of the acknowledgment block) and the sequence number immediately following the *right edge* (chronologically newest byte) of the block [23].

However, while selective acknowledgments can help to recover packet loss quickly, non-consecutive received segments still count towards the TCP receive window. Thus TCP cannot use information about non-sequential received segments to circumvent limits on data in the network imposed by the receiver and is at risk of throtteling the utilized bandwidth as a consequence of reaching receive window bounds.

### 2.3.5. Potential Issues of TCP

The Transmission Control Protocol has seen widespread use over the Internet. With a variety of optional extensions, and refinement of some of the core mechanisms and algorithms, it remains the most popular transmission protocol for reliable data communication over IP networks. Efficient and performant implementations of TCP are integrated into virtually all general purpose operating systems today. In the recent years, TCP has seen an even deeper integration into computer systems, through inclusion of special acceleration hardware within the NICs used in both general purpose computers and enterprise systems. Despite its popularity and tight integration, TCP also has a number of shortcomings depending on the precise requirements and target application area, some of which are illustrated through this section.

Because TCP has been standardized early in the continuous development and evolution of the Internet and was consequently used as a vehicle to study and explore various mechanisms in order to optimize reliable data transport over packet-switching networks such as provided through IP, its base specification does not reflect many of these research outcomes. While any TCP implementation should still be compatible with the base specification of RFC 793, TCP is extended through various standards or proposed standards, such as selective acknowledgments as described in RFC 2018. Furthermore, over time alternative algorithms for many of its core mechanisms, such as flow and congestion control, have been proposed to reflect various network characteristics. This implementation variety makes it difficult to build TCP implementations that achieve good performance over connections with other modern implementations.

Since TCP establishes reliable communication channels over fundamentally unreliable packet transports, without a reliable channel usable to bootstrap connections, it requires an elaborate scheme for connection management, as outlined in RFC 793 [20, pp. 10–12]. Noteworthy is TCP's three-way handshake to establish a connection between two sockets (passive-open socket referred to as *server*), which uses three messages (client $\xrightarrow{\texttt{SYN}}$ server; server $\xrightarrow{\texttt{SYN+ACK}}$ client; client $\xrightarrow{\texttt{ACK}}$ server) to establish a connection. Each `SYN` message synchronizes the initial sequence numbers with the respective remote end [20]. However, in practice, this exchange of messages also establishes that bidirectional communication is possible over the underlying path. A detailed figure outlining the possible TCP connection states and transitions is included as part of Appendix A. Any compliant implementation of TCP will have to implement this connection establishment and management logic, which can add a significant amount of complexity depending on the target system's architecture.

Finally, another significant issue is that of *head-of-line* blocking, induced by inherent mechanisms of TCP. When a sequence of data awaiting transmission is held up by the first datum, the transmission is said to be experiencing a form of head-of-line blocking. In computer networks, this phenomenon can occur at multiple layers, commonly when multiple independent data streams are preventing each other from transmitting data. Popular examples for this issue are queueing in packet switches, where incoming packets are processed in an in-order FIFO queue per incoming interface, or individual HyperText Transfer

Protocol (HTTP) requests sharing a single TCP connection and thus having to await the completion of any ongoing request. However, head-of-line blocking can also occur within a single data stream. For instance, if transmitted data are required to reliably arrive in order at the receiver, one might implement a so-called *stop-and-wait* ARQ protocol: for each transmitted message, the transmitter will wait for an acknowlegdment before sending the next message. If no acknowledgment is received within a certain time after transmission, the transmitter will issue a retransmission of the assumed to be lost message. This protocol can be said to constantly experience head-of-line blocking, as any pending data has to wait on the first datum in the queue.

TCP's ARQ mechanism is not employing a *stop-and-wait* scheme, but generally allows data to be injected into the network without regard of whether the first datum has been transmitted successfully. As outlined in Section 2.3.2, TCP flow control is a mechanism designed to avoid overwhelming the receiver by limiting the amount of unacknowledged data that can be injected into the network. However, the receive window concept fundamentally also introduces the risk of experiencing head-of-line blocking into TCP, as outlined through Figure 2.1. Whereas the TCP receive window, modeled as a sliding window, should not severely limit the utilized bandwidth in case of normal operation, in the face of loss events such a sliding window can limit the bounds of segments injected into the network until the lost segment has been acknowledged. Depending on the timeliness of loss event detection and corresponding retransmissions, the receive sliding window may not move forward for extended periods of time, even when selective acknowledgments as per RFC 2018 are employed (compare Figure 2.3). Even if the announced receive window is sufficiently large to never experience this particular issue, because the sender needs to retain any data starting from the first non-acknowledged segment in local memory, the receive window is effectively bounded by the minimum of the receive window size and the available memory at the sender. TCP selective acknowledgments potentially allow the sender to buffer data using a sparse data structure, holding only non-acknowledged segments, circumventing this issue while increasing the sender's logic complexity.

## 2.4. Alternative Transport Protocols

Apart from TCP, other protocols feature unique reliability constraints and flow control and congestion control characteristics. This section shall provide a brief overview of such alternative transport protocols.

### 2.4.1. Datagram Congestion Control Protocol

The Datagram Congestion Control Protocol (DCCP) is a message-oriented transport protocol. It has been developed out of the motivation to apply concepts established along TCP to traffic typically conveyed via User Datagram Protocol (UDP) because of a preference for timeliness over reliability, such as audio streams, Internet telephony, and online gaming [24]. UDP is a protocol conveying messages between a sending host and service

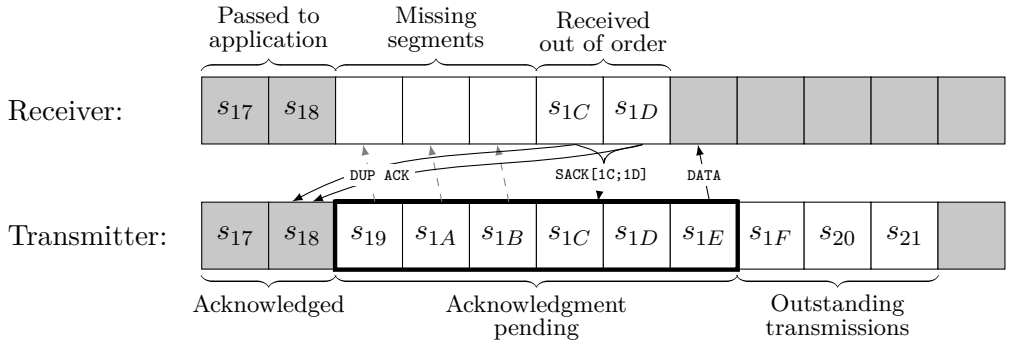



Figure 2.3.: TCP *receive sliding window* model in the face of lost segments. In this particular example, segments $s_{19}$, $s_{1A}$, and $s_{1B}$ have been either lost or reordered to arrive after segment $s_{1D}$. Depending on whether the TCP connection uses selective acknowledgments (`SACK`), the receiver will emit either such a message outlining the left and right edge of received segments, or a duplicate acknowledgment (`DUP ACK`) of the last sequential segment $s_{18}$ for each non-consecutive received segment. Note that duplicate acknowledgments cannot indicate how many segments have been lost, hence the sender may retransmit too few or too many. The receive window may only move forward once an acknowledgment for segment $s_{19}$ has been received.

identifier (port) to a receiving host and port. However, it does not establish the concept of a connection as defined with TCP and thus does not establish a relationship between any two messages, and consequently does not implement retransmissions or other mechanisms ensuring reliability. DCCP aims to supplement these characteristics with the concept of connections similar to those defined with TCP. It also adds congestion control mechanisms to ensure fair bandwidth utilization, for instance, when used along with other DCCP or TCP connections [24].

Based on this description, it can be concluded that DCCP is not of interest to this thesis. In fact, DCCP provides virtually no guarantees regarding reliable data transfer while retaining much of TCP's complexity, such as an elaborate connection management architecture.

### 2.4.2. QUIC

QUIC (originally an acronym for Quick UDP Internet Connections) is a general-purpose reliable transport protocol. It is similar to TCP in that it establishes a reliable transport service based on a fundamentally unreliable packet-oriented network protocol such as UDP over IP. However, QUIC attempts to improve over TCP, for instance, by embedding and mandating use of cryptographic protocols for encrypted and authenticated data transfer [5]. An additional explicit goal of QUIC is to eliminate head-of-line blocking as experienced in applications utilizing TCP.

However, it is important to distinguish between head-of-line blocking at multiple layers in the network, as explained previously. QUIC embeds the concept of streams, representing ordered sequences of bytes. Streams can be either bidirectional, meaning both endpoints can exchange data through a stream, or unidirectional, meaning a stream represents a directed transmission of a sequence of bytes from sender to transmitter. A single QUIC connection between two endpoints can contain multiple QUIC streams, where reliability guarantees, flow control, and congestion control are applied on a connection granularity [5]. This architecture is designed to circumvent the head-of-line blocking experienced, for instance, in HTTP: when a single connection is used to transfer multiple units of data (e.g., files or HTTP requests), for TCP, this means that a single loss event could potentially block the entire connection. However, by making the transport protocol aware of multiple independent data transfers, QUIC can effectively limit such head-of-line blocking to a single stream [5]. QUIC does not solve issues regarding head-of-line blocking in case of loss events within the data of a single stream.

Thus, while QUIC has many advantages over TCP as a transport protocol for its target applications, such as an underlying transport protocol for HTTP, it is not suitable to solve the issue of head-of-line blocking within a single stream of data. Furthermore, its additional complexity through mandated confidentiality and integrity protection and the streams concept make it infeasible as a transport protocol for this thesis.

### 2.4.3. Stream Control Transmission Protocol

The Stream Control Transmission Protocol (SCTP), specified through RFC 4960, is designed to address multiple issues it identifies within TCP. Especially relevant in the context of this thesis is that it recognizes that TCP only provides a reliable and in-order communication channel. Whereas some applications require both of these properties to be upheld, others only require either reliable or in-order delivery of data [25, p. 5]. Thus, SCTP specifies a transmission protocol capable of reliable and optionally in-order data delivery over an unreliable packet-oriented network protocol, such as IP. With the partial reliability extension of RFC 3758, reliable delivery can be optional on a protocol-chunk granularity [26]. Furthermore, it uses and adapts congestion control mechanisms specified for TCP through RFC 2581 [25, pp. 93–94].

In theory, the ability to ensure reliable unordered delivery on the transport protocol layer is ideal for this thesis. Such a mechanism can eliminate head-of-line blocking at the transport protocol layer, as the receiver does not have to wait for incoming data to be reassembled into an in-order data stream to provide it to the user application. The application can then further buffer this data using fast primary or secondary storage devices before reassembling it into an in-order stream. However, SCTP does not provide any information about the intended ordering of unordered transmissions (having an arbitrary stream sequence number) [25, pp. 88–89]. The protocol may reorder unordered transmissions as a consequence of retransmissions.

Furthermore, SCTP still retains the concept of a receive window for a flow control mechanism. Even unordered transmissions must respect this receive window [25, pp. 75–77]. Thus, SCTP generally retains the head-of-line blocking issues identified with TCP, although transport protocol reported receive buffer limitations are assumed to be rare given reordered transmissions of unordered messages do not prevent the receiver from handing off data to the application.

SCTP is also a rather complex protocol to implement. For these reasons, it does not appear to be a feasible protocol for this thesis. It does, however, establish valuable concepts for the problem of reliable transmission of measurement data.

# 3. System Model

The primary goal of this thesis is to design and evaluate system concepts specifically to transfer (stream) measurement data from a capturing device to a remote endpoint. This includes a survey of existing protocols suitable for this application and, depending on the results of this analysis, implementation and evaluation of a customized system architecture and protocol.

However, the target objectives such an architecture and protocol try to achieve and to which they are ultimately assessed can be vastly different depending on the precise application area. For instance, to transfer video or audio streams, it may be tolerable to occasionally lose or corrupt data fragments, as long as the overall quality of the transmission does not degrade substantially. In contrast, for text documents, this is most likely unacceptable. Thus, to enable comparing such an architecture and protocol to preexisting solutions as described in Chapter 2, engineer a new or modified solution, and finally evaluate its performance to the objectives set, it is essential to analyze and clearly state the assumptions and requirements this protocol must adhere to.

Given the vast space of potential application areas even within this thesis' explicit target of especially embedded measurement and data capturing devices, it seems reasonable to derive the requirements from a preexisting target device in hopes to generalize them to a reasonably large subspace of these applications. For the purposes of this thesis, this device will be a streaming time-to-digital converter. Such a device is useful in a large space of research areas and industrial settings, such as quantum research, fluorescence lifetime imaging, and light detection and ranging (LIDAR). Furthermore, it has interesting properties regarding the characteristics of its output data stream and the requirements imposed by the measurement applications utilizing this data, which will be discussed below.

## 3.1. Streaming Time-to-Digital Converters

Fundamentally, a time-to-digital converter (TDC) is an instrument to measure time intervals between two or more physical events. Conceptually these devices can be compared to an automatic stopwatch, triggered by electrical input signals. These devices convert a time interval $T = t_2 - t_1$, described through electrical impulses at time $t_1$ and time $t_2$ at the inputs, into a corresponding digital representation. Hardware TDC devices may feature multiple channels to simultaneously capture time intervals on multiple pulse streams or

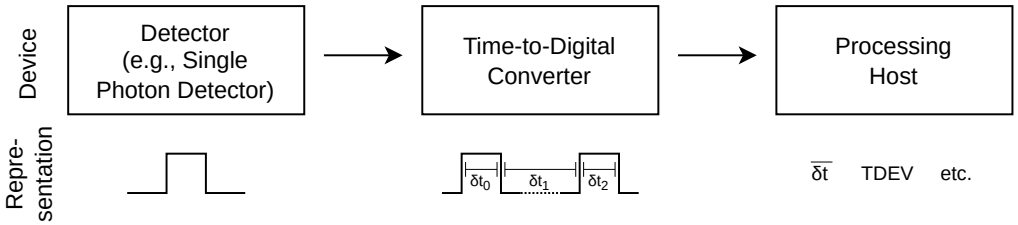



Figure 3.1.: Data representations within the Time Tagger measurement architecture. A detector outputs an electrical impulse, for which the Time Tagger captures the time differences between rising and falling edges. The attached computer performs further calculations on this data.

correlate impulses between different channels. Furthermore, these devices can integrate an advanced electrical frontend, for instance, to discretize an analog electrical input into a digital impulse based on a set trigger level [27].

A significant characteristic of TDC devices is their *dead time*, the smallest amount of time between the end of one measurement and the start of the subsequent measurement [27]. This parameter ultimately determines, depending on the application requirements, whether a device is limited to performing "one-shot" measurements or capable of operating on a continuous stream of input stimuli. Whereas devices with a comparatively large dead time are constrained to capturing within the bounds of a single measurement, potentially consisting of multiple time intervals, devices that feature a short dead time interval can be architected as "streaming" TDCs: instead of solely providing a limited number of time intervals in reference to a single start event $t_a$, such devices can continuously output time intervals between two subsequent events $\Delta t_e = t_e - t_{e-1}$, or as an offset to a joint timebase $t_e = t - t_{\text{ref}}$.

The TDC devices considered for the requirement analysis of this thesis constitute a family of streaming time-to-digital converters, with a significant part of their acquisition logic implemented within an FPGA. These devices can measure time intervals with a time jitter down to 4 ps over multiple input channels. Instead of correlating captured events directly, they assign a so-called *time tag* to the discretized detected impulses in reference to a shared timebase [28]. Thus, the output of these devices is a stream of time tags consisting of information about the captured event channel, the type of edge detected (rising or falling edge in the electrical signal), and a timestamp encoded as an offset between this event and the device-maintained timebase. They also emit certain status signals describing events within this data stream, such as reference timebase increments. A remote host is used to process this data further, either as soon as the data arrives or working on a recorded version of the data. Thus the TDCs are used purely as a data acquisition device. Figure 3.1 illustrates this processing architecture, outlining the different representations produced by the capture device (electrical impulses), TDC device (time difference measurements), and processing host (e.g., average time differences, time deviations, or other custom measurements).

## 3.2. Data Source Stream Characterization

The previously described family of TDCs allows deriving distinct characteristics describing the nature and behavior of the output data stream. Specifically, the device-internal interface width and maximum data rate, average and burst data rate, burst behavior, and additional interface characteristics as well as exposed metadata are of interest.

Fundamentally, the FPGA-internal interface of the TDC core is a streaming interface: the producer provides a data bus and a *valid* signal to indicate that a datum is present on the data bus. The producer expects the consumer to always be able to read and accept a datum on the bus. Each datum transmitted over the bus contains either a single measurement (time tag) or other status signals reported by the device. Thus, status and event signals are opaque within the data stream and do not require additional interfaces. However, the transferred status and event signals can be tightly coupled to the measurement data: for instance, an increment in the reference timebase directly relates to the previous and subsequent measurements and influences later interpretation of these data. This coupling ultimately mandates total ordering of the individual bus transactions to be maintained for the entire data transmission, even if the conveyed timestamps could otherwise reconstruct the ordering of individual measurements. An additional requirement to be inferred from this interface characteristic is that of losslessness. As a missing (erased) status or event signal can drastically impact the interpretation of measurement data, erasure of any such datum is unacceptable, in addition to any completeness requirements imposed by the individual application areas.

The provided data bus exposes a 32 bit-wide interface clocked at up to 750 MHz, resulting in a maximum theoretical data rate of 24 Gbit/s. This interface represents the combined time tag stream of all device channels. The TDC core employed in these devices can achieve this data rate in practice; there are no statistical guarantees regarding idle cycles where the bus does not hold a valid datum. Thus, to retain the usefulness of the TDC devices, the consumer of this interface must tolerate this data rate at least for short periods. For instance, when $n$ channels produce a time tag simultaneously, the consumer must be capable of accepting at least $n$ subsequent cycles of time tags to avoid disadvantaging any input channel's transmission channel imposed dead time over another's. Demanding measurement applications may utilize the entire available bandwidth of the TDC core. Thus any transmission protocol implementation should, within reasonable bounds, temporarily tolerate such data rates even if they exceed the available external link bandwidth.

In the event that the data processing pipeline within the TDC device can no longer keep up with the current data rate and data loss is inevitable if the data source continues producing data at that rate, the TDC core can be informed of this situation. The core will then halt the acquisition of new data in an effort to transfer as many valid measurements within the core to the remote host, before finally reporting this *overflow* condition. A transmission protocol implementation must thus predict such a condition and still be able to accept the remaining valid measurements contained within the core.

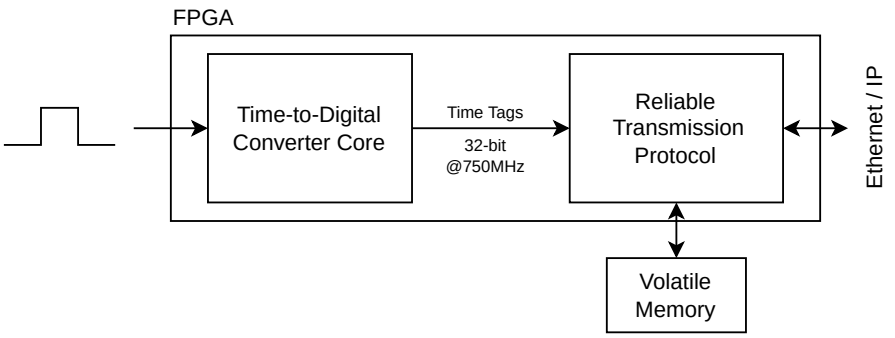



Figure 3.2.: Illustration of the reliable transmission protocol's system context based on a TDC device.

Figure 3.2 provides an illustration of this system context. The class of devices outlined within Section 3.1 features interesting properties concerning their architecture, data stream characteristics, and application areas. Nonetheless, these devices are not alone in facing the problem of transporting captured measurement data to an end host, with strict requirements for the imposed data rate and data stream reliability. Furthermore, the analysis of this section concludes that these devices feature a rather generic interface, which can also apply to different applications. Related classes of measurement devices such as ADCs or networked oscilloscopes, or even application domains such as video capturing systems, are suspected to have a very similar set of requirements and interfaces.

## 3.3. Target System Architecture

Apart from the measurement application and the provided interfaces, it is also essential to consider the larger system context in which the aforementioned issues shall be solved. Specifically, for this thesis, the developed solution must be implemented within hardware systems such as FPGAs or ASICs. While being practical platforms for developing measurement devices such as the ones described above, these systems bring their own sets of unique challenges, particularly when integrating into computer networks and interoperating with general-purpose computers. This section will thus present some of these unique challenges to better understand the design space and the problem at hand.

### 3.3.1. Hardware Ethernet and IP Interfaces

To properly analyze preexisting transport protocols for their applicability to the target application area and construct transport protocols efficiently representable within a target system, it is crucial to consider their compatibility with the fundamental interfaces any such protocol must integrate. For the intents and purposes of this thesis, the transport protocol has to be integrated with the target system's interfaces for data acquisition, Ethernet & IP communication, and a volatile memory device for any required data buffering.

Hardware-based interfaces for data communication are typically considerably different in their basic architecture compared to software-controlled interfaces of general purposes computers. Transmission and reception of network packets within software systems generally utilize copies of entire packets within the system's main memory, written to or read from this memory through the NIC. In contrast, hardware or FPGA-based systems more commonly utilize stream-based interfaces to process data: in such systems, data is transferred and processed through fixed signal paths, $n$ bit at a time, at a fixed rate (clock frequency). So-called *pipeline stages* divide these paths, where each stage employs registers to hold input data for the duration of a clock cycle, providing it as input signals to combinational logic determining output signals, which in turn are connected to the inputs of registers of the subsequent pipeline stage. The input clock frequency of stream-based and pipelined processing stages determine the data processing rate; in theory, they can be clocked at a rate sufficient to process incoming or synthesize outgoing data as fast as it arrives or must be sent over the transmission channel. This can eliminate the need to buffer entire packets prior to transmission or processing and thus reduces the implementation's memory requirements. Furthermore, a pipelined architecture allows implemented logic of the individual stages to be fully utilized in every clock cycle as new data enters the pipelined processing architecture. This is in contrast to a general-purpose processor, for which one generally cannot make any strict timing guarantees concerning the number of cycles to process incoming or outgoing packets. Thus it requires incoming and outgoing packets to be resident or assembled in memory. However, performing random-access reads and permutations of packet data in a general-purpose processor system enables the implementation of elaborate logic, which would not be possible in an unbuffered stream-based architecture. Finally, depending on the data source or sink, stalling the processed data stream might not be possible; in this case, each pipeline stage must take a constant maximum processing time ($n$ cycles) for its entire logic, regardless of the processed data.

The LiteEth FPGA MAC and IP core's architecture is based on two pipelined data streams per physical interface for receive and transmit paths. The MAC core integrates with the Ethernet physical layer (PHY) via a Media Independent Interface (MII) such as the 10 Gigabit Media Independent Interface (XGMII). The core's receive side consumes protocol headers and collects contained information to selectively connect the decapsulated (*depacketized*) data stream onward to further processing stages. This architecture works because Ethernet frames and IP packets contain all information required to determine the next processing stage within their header section, preceding any payload data. Because the receiver cannot control the rate at which data arrives, all pipeline stages must accept new data every clock cycle; they cannot stall the reception unless incoming data is buffered sufficiently. Analog to the receive side, packet transmission stages utilize metadata provided by the data source to synthesize appropriate protocol headers, followed by payload data accepted from the source. Once a transmission begins, sources must provide valid data in each clock cycle for the remainder of the packet, as the unbuffered pipeline architecture will transmit any data immediately through the outgoing interface.

While it is possible to utilize buffers for outgoing and incoming packet data to perform arbitrary multi-cycle operations on the transmit or received data, the requirement of allocating sufficiently large buffers means that an unbuffered pipelined streaming architecture is strongly preferable wherever feasible. Furthermore, processing data as fast as it arrives

or is transmitted (at *line rate*), guaranteed for an unbuffered non-stalling stream processing pipeline, is preferred over multi-cycle operations requiring parallelism in the processing logic to saturate available link bandwidths. TCP's non-interleaved sequential data transfer characteristics, with a header preceding each segment's data, map well to such an architecture. Nonetheless, TCP's multitude of optional extensions and variable-length header make encapsulation and decapsulation of TCP segments non-trivial. Generally, extensible protocols and ones with a more complex data structure and encoding will require more logic to implement in a streaming architecture.

### 3.3.2. Pending and Retransmit Data Buffers

While the storage architecture within general purposes computers generally features some main memory for volatile storage, accessible to software through a unified address space, purpose-designed hardware platforms and FPGAs alike do not have such a fixed memory architecture. Instead, they provide more direct control over the memory primitives available within the system [29]. The storage architecture, along with particular attributes such as storage size, power consumption, random-access capability and performance penalties, interface data width, and access latency, can be essential in choosing an appropriate storage medium and influence the feasibility and complexity of transport protocol implementations.

Fundamentally, FPGAs offer different types of integrated memory and can interface with external memory devices. Internally, FPGAs can utilize flip-flop registers as a form of distributed memory across the logic fabric referred to as distributed random access memory (RAM), allowing writes synchronous to a clock signal, with a latency of one cycle, and asynchronous access to stored data. Furthermore, block RAMs are larger, dedicated arrays of flip-flop memory cells that allow synchronous read and write access to stored data, with an access latency ranging from one to two cycles depending on the target platform. While such on-chip static random access memory (SRAM) based memory cells have low access latencies, they are generally less dense, more expensive, and have significant power consumption; it is common for hardware or FPGA systems to utilize dynamic random access memory (DRAM) for volatile bulk data storage. However, DRAM memories commonly require complex logic to interface with and have significant and varying access latency, depending on the data access pattern.

To establish a reliable communication channel over one which does not meet the criterion of losslessness, all transmitted but not yet acknowledged data must be stored at the sender for potential retransmission. Assuming reasonable network path characteristics for this thesis' target application area of a 10 Gbit/s link with $\leq$ 5 ms RTT latency up to the receiving application, this results in a bandwidth delay product of up to 51.2 Mbit, which needs to be stored by the sender if the link bandwidth is to be fully utilized, excluding any untransmitted data. While reasonably modern and large FPGAs may collectively feature sufficient block RAM and distributed RAM to accommodate this data, utilizing such amounts of SRAM is expensive and would occupy valuable resources, preventing usage by other logic implemented on the FPGA. Thus using DRAM to buffer transmitted data

is preferential for this thesis. The increased latency and controller complexity of DRAM needs to be considered in the protocol design, for instance, by reducing the amount of blocking read and write operations required.

Another essential aspect to consider in the design of transport protocols, depending on and influencing the choice of buffer memory, is that of memory layout, particularly address and data (word) widths. Because of address width requirements and interface complexity and efficiency, it is generally infeasible to have bit-addressable memory devices. Instead, data is accessed in words of $n = 2^m$ bit, $m \in \mathbb{N}_0$. When memory is accessed from logic running at a low clock frequency it may be provided with an interface aggregating multiple memory words to saturate available memory bandwidth. Thus, utilizing the entire available memory bandwidth from a synchronous logic domain (clock domain) $C$ constraints the *effective* memory word width to be a function of the maximum of the memory's physical (interface-determined) word width $N$ and $M = \lceil \frac{B}{f_C} / N \rceil \cdot N$, $B$ being the memory bandwidth and $f_C$ the clock domain's frequency. Depending on data alignment and the given transport protocol's fundamental word size, this can directly impact the processing logic complexity, as further outlined below.

### 3.3.3. Hardware Limitations

To design a protocol that is efficiently representable through FPGAs or ASICs, various other characteristics and limitations can be considered. While the previous sections focused on specific aspects of given protocol implementations, the following will outline some general limitations of hardware platforms and their influence on the protocol design space.

Especially in complex combinational logic circuits implemented within FPGAs, the time it takes for inputs to fully propagate through logic elements and provide stable output signals can be of concern. For instance, processing a 10 Gbit/s data stream one bit at a time would require a clock frequency of 10 GHz, requiring all combinational logic to fully propagate within 100 ps. Modern FPGA platforms can support clock frequencies of approximately 800 MHz for low complexity logic. Because handling protocols such as IP generally requires relatively complex logic, it is common to process such data streams $n = 2^m$ bit, $m \in \mathbb{N}_0$ at a time, which reduces clock frequency requirements by a factor of $n$. For instance, 10 Gbit/s Ethernet can be processed at 64 bit/cycle, requiring the logic to run at 156.25 MHz. However, as a consequence, depending on the protocol specifics, this may require more complex logic to process incoming or outgoing data: for instance, a byte-aligned protocol header may be located at an arbitrary byte offset within a 64 bit bus word. If these implementation-specific constraints are known ahead of time, appropriate alignment of protocol headers and transmission protocol words can help to reduce the protocol's implementation complexity. Otherwise, unaligned access to protocol data needs to be implemented.

# 4. Problem Statement

Chapter 2 establishes essential background knowledge to understand the problem of providing a reliable communication channel using a fundamentally unreliable packet transport, such as the IP protocol. It further describes TCP as an example of a protocol providing a reliable transport service using IP. It discusses many important aspects such as automatic repeat requests (ARQs), congestion control, and flow control.

However, considering the system model as illustrated in Chapter 3, there are many potential issues in using TCP or other transport protocols for reliable transmission of measurement data. This is especially true for an implementation operating at the target data rate of 10 Gbit/s and for one implemented through logic synthesized for hardware (FPGA or ASIC) devices. These problems include bounds on the implementation complexity and timing characteristics, TCP's extensible nature and variable-length header, and its complex connection management architecture. However, most prevalent is the problem of head-of-line blocking, inherently caused by the flow control mechanism and the data structures TCP expects to be maintained at the transmitter. All of these issues are in contrast to a hardware implementation of such a reliable transmission protocol, making efficient use of available resources.

Therefore, this thesis will analyze different aspects of reliable transmission protocols and devise concepts and algorithms suitable for efficient implementation within the systems described throughout the system model. Specifically, this thesis will design a memory management architecture to maintain pending, in-transit, and to be retransmit data at the transmitter efficiently. Based on this component, a network protocol for efficient transfer and reliable in-order delivery of measurement data shall be designed.

To validate the designed system concepts, a proof-of-concept implementation of the memory management architecture and developed network protocol shall be provided. This proof-of-concept implementation can be used to determine whether the developed system concepts are correct, applicable the problem at hand, as well as adhering to the performance requirements as illustrated in Chapter 3.

# 5. Design

Based on the related work and the system model elaborated in Chapter 2 and Chapter 3, and following the problem statement of Chapter 4, this chapter will discuss key design aspects of the system architecture and transport protocol composing HELIX. The particular design of HELIX is based on and influenced by many of the restrictions and requirements of the target application domain, hardware constraints, and other aspects elaborated within the system model. Furthermore, HELIX utilizes many concepts and underlying techniques as analyzed in the related work outlined in Chapter 2. Nonetheless, the exact composition of preexisting concepts and techniques, together with novel design approaches, are key to designing a formal transport protocol and system architecture that is correct, efficient, and applicable to the target application domain.

This chapter will thus focus on developing and discussing individual components influencing the overall design of HELIX, such as the in-transit and retransmit data management architecture, integration of congestion and flow control mechanisms, and receiver host behavior. Finally, these components are combined to provide a coherent model of the developed transport protocol.

## 5.1. In-transit and Retransmit Data Management Architecture

As concluded based on the findings of Section 2.1, it is vital for any implementation of a reliable communication channel as defined for this thesis to store all transmitted but not yet acknowledged data, given that intact arrival at the receiver is not guaranteed. The explorations of TCP and other protocols within Chapter 2 have shown that the methods and structures used to store this data can be closely related to the overall transport protocol, specifically the feasibility and complexity of advanced mechanisms such as selective acknowledgments, and flow control and congestion control. Even more so, any restrictions in memory size, random-access capabilities, access granularity and latencies, and other aspects considered within Section 3.3.2 may significantly limit the design space of such mechanisms, especially when considering the implementation complexity involved in overcoming memory restrictions. For example, a non random-access capable memory device storing in-transit data is in fundamental conflict with an efficient implementation of selective data retransmissions. With the explicit goal of eliminating any head-of-line blocking caused by the transmission protocol, the mechanisms holding data presumed to be in-transit and data presumed to be lost and thus having to be retransmit are crucial to the design of such a protocol. This section will illustrate issues in unifying the technical challenges presented in Section 3.3 with data management architectures demanded by

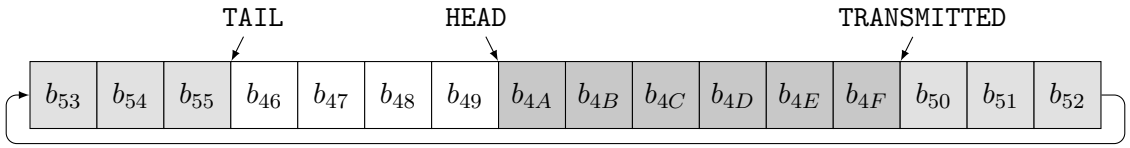



Figure 5.1.: Circular buffer representing outstanding and transmitted bytes in a TCP data stream. The fixed-length contiguous memory region logically wraps around from the last to the first index. The `HEAD` and `TAIL` pointers indicate the contiguous and potentially wrapping region of valid data in the buffer; the `TRANSMITTED` pointer divides this region into a section of bytes injected into the network and a range of outstanding, to be transmitted data.

TCP and restrictions imposed by this thesis. It develops and presents an architecture for managing in-transit and retransmit data, designed to integrate with DRAM interfaces in hardware, avoiding head-of-line blocking, but exposing certain restrictions that ultimately make it incompatible with preexisting transport protocols.

### 5.1.1. Non-sparse Circular Buffers

A conceptually intuitive and straightforward way to represent any untransmitted, in-transit, and to be retransmit data, mapping particularly well to the requirements and limitations of TCP, is a queue represented through a circular buffer. In such a data structure, data is written sequentially in memory, with the end of the (virtually) contiguous memory region wrapping around to the start of the memory region. In such a memory structure, the start and end of the contiguous region containing valid data are typically indicated through two pointers, pointing to the start (head) and end (tail) offsets of this region, respectively, in addition to a Boolean flag tracking—depending on the implementation—whether the buffer is empty or full. In the case of TCP, a third pointer would separate this memory region into a first part containing transmitted but not acknowledged data and a second part containing untransmitted data. Figure 5.1 shows an illustration of such a data structure. Of importance for data structures tracking in-network data is the transport protocol's basic word size: for instance, TCP works on a byte granularity and can transmit frames containing one or more bytes. Thus, any data structure buffering TCP data needs to keep track of transmitted data on a byte granularity as well.

Such a circular buffer queue is trivial to represent in both software-based implementations and hardware or FPGA logic designs. When using an additional variable to store the head element's byte offset in the overall data stream, and thus being able to calculate every buffer element's TCP sequence number based on this variable and the offset from the head element, this data structure is sufficient to model a TCP stream's payload data at the transmitter, following the base TCP specification of RFC 793. Furthermore, when not accounting for any unused space in the circular buffer contiguous memory region allocation,

it has a remarkably low memory overhead, given that it can utilize the entire buffer allocation for payload data without any additional metadata or alignment constraints.

In addition to that, it is possible to implement an efficient selective retransmission mechanism using such a data structure. Because a receiver issuing selective acknowledgments indicates the sequence number offsets of a data region to acknowledge, combined with the sequence number offset of the circular buffer's head element, an implementation can use this information to directly index into the buffer at known offsets and retransmit any unacknowledged data in front of the acknowledged data. However, the issue of head-of-line blocking at the transmitter remains: without introducing of more complex management data and logic, potentially unbounded in size and complexity, such a sequential data structure cannot arbitrarily reuse memory occupied by selectively acknowledged data for new, outstanding, and to be transmit data.

The Linux kernel's TCP implementation uses a variation of this data structure. In order to avoid copying payload data throughout the kernel's network stack, it uses a circular buffer of pointers to chunks of data, so-called `sk_buff` data structures, allocated elsewhere in memory. This additional layer of indirection over such a unified data structure enables the kernel to effectively achieve a TCP transmission queue representation equivalent to the circular buffer presented here, but chunking data into TCP segments prior to enqueuing them in the circular transmission buffer. Consequently, to schedule transmission of any given segment, the implementation simply removes this pointer from the queue and provides it to the IP layer without copying any proper payload data [30]. However, in hardware systems employing a pipelined streaming architecture, such an additional layer of indirection can actually increase logic complexity and access latency. Nonetheless, if dynamic reordering of elements in a queue is required, operating on pointers to memory regions can be favorable compared to relocating payload data in memory. Finally, such a data structure—given it is no longer contained in a single, consecutive memory allocation—introduces the additional challenge of allocation of chunks in memory along with memory fragmentation and related issues.

### 5.1.2. Sparse Buffer Data Structures

Fundamentally, to avoid the aforementioned issue of head-of-line blocking caused by the data structure representing pending and in-network data at the transmitter, a sparse data structure representing only unacknowledged data of the transport protocol's data stream is required. Such a data structure must be able to reclaim and reuse memory occupied by acknowledged data within a data series and track or handle such gaps in a series without imposing excessive overhead in memory usage and logic complexity. Additionally, any selective acknowledgments received must be handled efficiently, implying that locating data to be retransmitted and reclaiming memory containing acknowledged data must be operations with reasonable logic complexity, timing bounds, and limited memory bandwidth utilization.

The circular buffer queue structure described in the previous section is based on an array: the ordering of individual data composing a transport stream's data series is encoded in the index of data stored in the buffer. While such a structure is efficient at accessing data at known or calculated offsets, with a complexity of $\mathcal{O}(1)$, it does not offer an efficient mechanism to represent gaps in a data series and allow this memory to be repurposed for new out-of-sequence data. Removing $k$ consecutive elements out of $j$ elements of an array-based circular buffer at index $i \in [0; j)$ requires $\min(i, j - i - k)$ elements to be moved in order to retain a consecutive sequence of elements, where each move consists of a read and write operation, and hence generally has linear complexity $\mathcal{O}(n)$ [31].

A layer of indirection in the transmit queue, such as employed by Linux for TCP and described in the previous section, can reduce the amount of data to be moved: for a circular buffer queue of chunks (e.g., `sk_buff`s) of data, if data removed from the queue align with the boundaries of their respective chunk containers, it is sufficient to remove these chunk pointers from the circular buffer, without relocating any actual payload data. This reduces the amount of read and write operations required to retain a consecutive sequence of pointers to chunks by a factor of $n = \text{avg}(\text{datum}/\text{chunk})$.

Linked lists are data structures that, in contrast to array-based data structures, support efficient $\mathcal{O}(1)$ insertion or removal of elements at the head, tail, or in between elements, provided that the required pointers for these operations are available [31]. In contrast to array-based data structures, offsets of elements generally cannot be used to index into linked list data structures. Instead, locating a particular element requires traversal from a known element (e.g., the list's head element) and thus is an operation of linear complexity $\mathcal{O}(n)$. Because linked list nodes are not necessarily located consecutively in memory, they also require allocation mechanisms and must handle memory fragmentation.

An issue shared by both sparse data structures presented above is the mapping of transport protocol stream offsets, used throughout the transport protocol to reference specific data or ranges of data (e.g., TCP sequence numbers), and the location in memory where these data are stored. For example, when pointers to acknowledged data chunks in an indirect circular buffer queue have been removed, the logical index of a given segment in the transport protocol stream does not necessarily correspond with the index of the chunk containing the segment's data any longer. Instead, each queue entry would have to indicate the stream offset it represents, and locating the desired elements would require a binary search operation of $\mathcal{O}(\log n)$ complexity. For a linked list without prior knowledge about the node's location in memory, both sparse and non-sparse lists require a linear search of $\mathcal{O}(n)$. Thus, even though chunk removal operations without relocation of payload data are possible with these data structures, locating data based on a transport protocol stream offset is fundamentally less efficient compared to a non-sparse direct circular buffer queue.

Given that selective retransmissions inherently require indexing into buffered data based on transport protocol stream offsets, a modification to the indirect circular buffer queue model can allow data to be located with $\mathcal{O}(1)$ complexity: if the array-based circular buffer, containing pointers to chunks of data, is not modified to retain a consecutive sequence of elements, but instead each pointer can be marked as invalid, the mapping between

circular buffer indices and data chunks does not change when gaps are introduced in the retained data and memory occupied by acknowledged chunks is reclaimed. However, even if this first mapping does not change, such a data structure does not necessarily allow a direct translation of transport protocol stream offsets to memory locations: only if the data chunks are of the same size (all containing $n$ data words), a word index $i_w$ can be translated to a memory address $a_w$ as $a_w = M[\lfloor (i_w - o)/n \rfloor] + (i_w \mod n)$, with $o$ being the current head element offset in the range of transport protocol stream indices and $M$ the array-based circular buffer queue mapping to data chunks. This additional restriction of every chunk containing $n$ data words may conflict with timely delivery of data: to ensure efficient removal operations on either indirect circular buffer queues or linked lists, the size of chunks or list nodes respectively should correlate with the size of segments transmitted. This is because loss or reordering events during transmission fundamentally affect entire packets, and removal operations in either data structure are efficient if and only if to be removed data ranges align with the boundaries of chunks or list nodes. Consequently, before a given fixed-size segment can be transferred from transmitter to receiver, sufficient data must be collected for its transmission. While timely delivery is not an explicit requirement for this thesis, it must still be possible to transfer data even if the total number of words to transmit is not evenly divisible by the number of words in a segment, given that the data source cannot be influenced to adhere to this requirement by the transport protocol.

### 5.1.3. Linked Fragmented Buffer

The previous section's exploration of potential in-memory representations of pending, in-transit, and to be retransmit data appears to suggest that there is a fundamental tradeoff between the ability to efficiently access data at known offsets in a data series (such as provided in TCP through sequence numbers), the ability to transmit segments of varying lengths, and the capability of such data structures to reclaim memory occupied by acknowledged data ranges efficiently. Nevertheless, up to this point, only self-contained data structures were considered, and how such data structures would be capable of operating based on information provided by existing transport protocols such as TCP. These structures are inherently *transparent* to remote hosts, meaning that the receiver has no information about how data is managed in the transmitter's memory. This implies that the transmitter must translate the protocol-provided information, such as sequence numbers, into internal memory locations. With this thesis' basic presumption of being able to provide an efficient transport protocol purpose-built for hardware and FPGA platforms, it appears viable to explore data structures influencing the protocol design and inherently cooperating with the receiving host to overcome limitations as explored previously.

To that end, this thesis proposes a so-called *linked fragmented buffer* as a data structure and set of mechanisms to model pending, in-transit, and to be retransmitted data of a transport protocol. This memory model is based on a singly linked list with specific properties, making accesses into and manipulations of the list's structure as required by the transport protocol possible with bounded logic and timing complexity. At the same time, this memory model imposes certain restrictions and assumptions on the employed trans-

port protocol and requires the cooperation of the receiving host to ensure correct operation concerning the formal transmission model by the sender. This section first elaborates on the ideas and mechanisms of the linked fragmented buffer and the aforementioned limitations. After the fundamental concepts of this memory model have been established, the section continues by describing the implementation of individual operations required for reliable transport protocols. Finally, it discusses some further questions and challenges in relation to implementing the developed data structure on an FPGA platform and other mechanisms employed to ensure a reliable integration with transport protocols.

#### 5.1.3.1. Development of the Data Structure

Fundamentally, a data structure modeling the transport protocol's data stream must have the following properties and support the following operations, respectively:

- It must be able to hold data placed on the underlying transmission channel (in-transit) and data awaiting transmission (pending).
- To accept new incoming data, the data structure must be capable of appending new data to the end of the pending data series.
- It must support efficient transitioning of the pending data series' head element to the in-transit state to reflect transmissions over the underlying transport medium.
- Data known or suspected to be lost during transmission requires eventual retransmission. Thus, the data structure must support marking ranges of data as *lost* and allow an implementation to retrieve these data and schedule them for retransmission.
- It should provide mechanisms to retrieve metadata of the oldest non-acknowledged data chunk to enable the integration of automatic repeat request (ARQ) mechanisms. For example, an ARQ mechanism might utilize the original transmission timestamp of a given data chunk to determine whether retransmission is necessary.

The most straightforward approach to using a singly linked list for modeling in-transit and pending data of a reliable transport protocol data stream is to use a word-addressable memory region and retain a pointer to the first inserted (list head) element (node) in the chain. By having each list node contain metadata about whether there is a successor node and the address of the successor node, as well as some payload data, retaining a pointer to the first node enables reaching all other nodes in the chain. To distinguish in-transit from pending data, a second pointer to the first pending node is retained as well. This basic structure already introduces an inherent restriction: to avoid manipulating any list node's payload data while transitioning a given node from the pending to in-transit state, any pending node's payload data length must align with the transport protocol's segment lengths. Thus, this data structure requires the stream-based data sink to be mapped onto a discrete, packet-based structure not just when data is transferred over the underlying packet-based communication channel but as soon as it is enqueued and scheduled for future transmission.

Such a structure is in itself rather inefficient for appending new pending data: the insertion of new data requires traversal of the entire list, starting from the pointer to the first pending

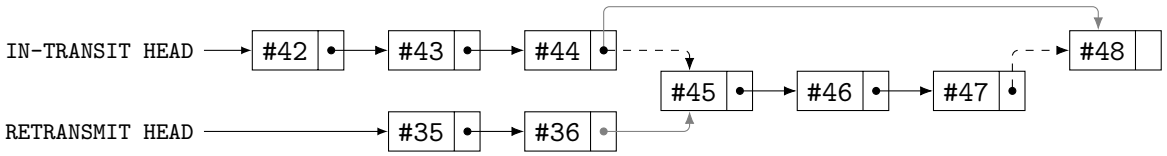



Figure 5.2.: Abstract representation of a linked list data structure maintaining list nodes holding transport stream payload data. Labels of nodes are monotonically increasing at insertion in the primary (in-transit / pending) list. This figure, in particular, shows how ranges of list nodes can be moved to a second, retransmit list through a constant number of pointer modifications, and with constant complexity $\mathcal{O}(1)$, assuming all required pointers are known beforehand. Dashed arrows indicate pointer references to be removed; gray arrows indicate new pointer values.

node, an operation having linear $\mathcal{O}(n)$ complexity. This can be reduced to a constant-time $\mathcal{O}(1)$ operation if the last node's pointer is stored in addition to the start pointers of the in-transit and pending segments. Other operations are generally efficiently implementable, such as transitioning a node from the pending to the in-transit state. For this operation, the pending segment's start pointer has to be updated to reflect the current pending segment's head node successor, which requires a single memory read operation of $\mathcal{O}(1)$.

As stated above, the segmented linked list data structure presented here must further be able to mark ranges of data as being *lost* and allow a surrounding transport protocol to retrieve this data for eventual retransmission. To avoid head-of-line blocking, this mechanism must fundamentally work with selectively acknowledged sections of data; hence multiple distinct ranges in the data might be marked as lost at any given time. Because of the previously set requirement that payload data of list nodes correlate to segments of the transport protocol, and loss or reordering events fundamentally affect only entire network packets, it can be assumed that *lost* data ranges on a list node granularity. A viable approach to implement the concept of *lost* data could be to introduce an additional metadata attribute per list node, reflecting this marking. However, to locate such *lost*-marked list nodes for eventual retransmission requires a linear $\mathcal{O}(n)$ search through the in-transit nodes segment of the linked list structure. A better strategy appears to be utilizing the fact that removing and appending a series of nodes in a singly linked list are operations of $\mathcal{O}(1)$: without actually relocating any list nodes or payload data in memory, by manipulation of a constant number of list pointers alone, a series of *lost* list nodes can be removed from the in-transit list and appended to a second *retransmit* list. This process is further illustrated in Figure 5.2. Maintaining a second, independent list structure of *lost* and to be retransmit nodes ensures that locating data for retransmission is a constant time operation, as thehead-pointer of the *retransmit* list always references the oldest *lost* node.

Still, as shown in Section 5.1.2, a linked list data structure does not solve the issue of mapping transport protocol stream offsets to the location of data stored in memory. Even if all list nodes were to have the same payload length, because of the non-contiguous allocation nature of a linked list and the list's structure being maintained through pointers located in

list nodes, resolving a stream offset to a memory address fundamentally requires a linear search of $\mathcal{O}(n)$ over the list nodes from a given starting node. Furthermore, the mechanism to separate *lost* from *in-transit* nodes in constant time, as presented above, also requires prior knowledge of the respective list nodes' addresses to be transitioned to the *retransmit* list, as well as knowledge of the predecessor node's address. Such an efficient mapping, however, is an essential requirement for effective processing of selective acknowledgments reported by the receiver: in contrast to cumulative acknowledgments such as employed by TCP as specified through RFC 793, selective acknowledgments—at their core—can acknowledge ranges of data which are not necessarily consecutive to the transmitter's in-transit head data element, with the acknowledgment range bounds (left and right edge of the acknowledgment) represented by offsets within the transport protocol's data stream (e.g., TCP sequence numbers). Thus, as self-contained data structures, singly-linked lists appear to be subpar representations for in-transit and pending data of a reliable transport protocol compared to array-based data structures.

However, when considering the integration of specific inherent properties of the list structure with the transport protocol itself, expensive linear searches over the list in response to selective acknowledgments can be avoided entirely. Enabling this mechanism is the fact that, in contrast to array-based data structures, changing the structure of a linked list, maintained through pointer-relations in each list nodes' metadata, does not require relocation of any list nodes in memory. This property implies that reclaiming memory occupied by acknowledged list nodes by removing these nodes from the in-transit linked list is possible without affecting the memory locations of unacknowledged, in-transit list nodes. Because (i) selective acknowledgments are acknowledging ranges of data that arrived intact at the receiver, (ii) the memory location of buffered data at the transmitter can be determined before transmission of this data, and (iii) the fact that this memory location does not change when modifying the in-transit list, it is feasible to attach the memory address of a given segment's transmitter-maintained copy of data to outgoing transport protocol segments during transmission. By having the receiver report the memory addresses of the leftmost and rightmost segments of a selective acknowledgment back to the transmitter, the transmitter can utilize these addresses for the transformation of its linked list memory structures directly, without performing any linear search operations on the memory.

Because removal of elements in a singly linked list requires knowledge of the preceding element's pointer, in addition to pointers of the leftmost and rightmost acknowledged list elements as described above, this pointer must be provided within acknowledgments as well. However, this poses an issue with selective acknowledgments: when selectively acknowledging a non-consecutive segment, the receiver does not know of the preceding segment's memory location at the transmitter. To circumvent this issue, the pointer to the respective preceding element is included along with every transmitted segment, such that the receiver always has sufficient information to issue selective acknowledgments for arbitrary ranges of received segments.

This basic singly linked list data structure, along with any restrictions and transport protocol interactions elaborated in this section, appears to be a viable representation for buffering a reliable transport protocol's pending, in-transit, and retransmit data, given the

Control data:

| | | | |
|---|---|---|---|
| In-transit Data Head: | 1C00h | Retransmit Data Head: | 0800h |
| Pending Data Head: | 0400h | Retransmit Data Tail: | 0C00h |
| Pending Data Tail: | 1000h | Retransmit Chunks: | 2 |
| In-transit / Pending Chunks: | 2 / 2 | Next Free Chunk: | 1400h |

DRAM memory layout:

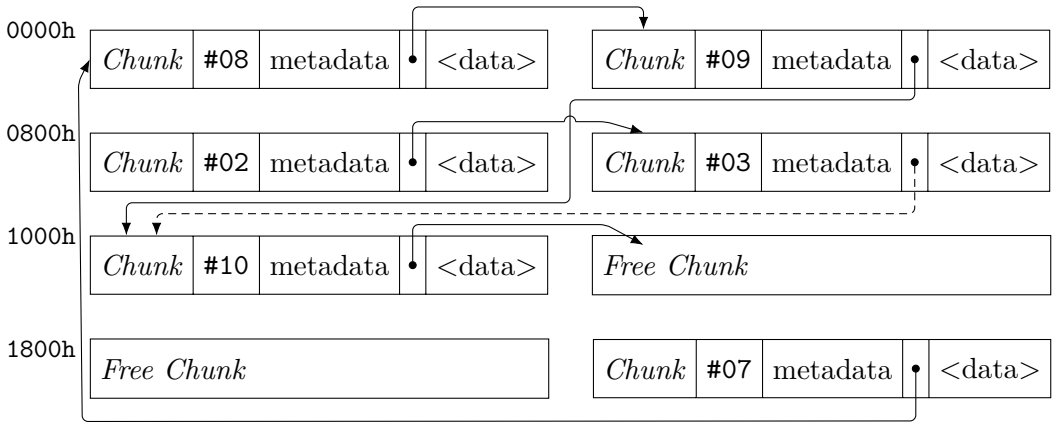



Figure 5.3.: Depiction of the in-memory layout of a linked fragmented buffer and required control data maintained in device-internal registers. Solid arrows indicate pointer relations established in memory; dashed arrows indicate invalidated pointer relations through the control data.

specific requirements imposed by this thesis. Figure 5.3 further outlines the structure of the linked fragmented buffer through a concrete example showing how different list nodes (*chunks*) and their pointer relations, in conjunction with control data registers, compose the pending / in-transit and the retransmit linked lists. The particular example of Figure 5.3 uses fixed-size chunks of 1024 data words. Each chunk is assigned a monotonically increasing identification number upon insertion. Pointer relations maintained within the chunks themselves, combined with control data registers, compose two disconnected linked lists: in this example, the pending and in-transit list spans from chunk #07 to chunk #10, with the *pending data head* pointer separating the list into an in-transit segment and a pending segment, starting at and including chunk #09. The retransmit linked list spans from chunk #02 to #03. This linked list has been split from the in-transit / pending list without rewriting or relocating the chunks in memory. Hence chunk #03 still contains a pointer to address `1000h`; however this pointer is invalidated through the *retransmit data tail* control register. Section 5.4 will further integrate this in-memory structure into a transmission protocol, outlining exchanged messages and control data.

#### 5.1.3.2. Linked Fragmented Buffer Operations

Based on the optimized data structure as described in the previous section and Figure 5.3, the operations on this data structure as required by interactions with the transport protocol shall be elaborated:

**Insertion of pending data** First and foremost, it must be possible to insert new pending data into the transport protocol buffer. Inserting new data into the pending and in-transit linked list involves appending a new node, referred to as a chunk, to the tail of this list. Given that the tail pointer of this list is maintained within the control data registers, the next-chunk pointer of this tail element can be rewritten to point to the inserted element directly. An issue not solved as part of this data structure's design, and depending on the precise memory layout and chunk sizes chosen for this data structure, is that of memory allocation of inserted list nodes. Strategies for memory allocation are further discussed within Section 5.1.3.4. Assuming sufficient space has been allocated, inserting a new chunk into the list involves writing a header to retain the linked-list structure and writing the chunk's payload data. As discussed in Section 5.1.3.1, the amount of data written must not exceed the transport protocol's maximum segment size, as chunks in the linked fragmented buffer structure should correspond to segments of the transport protocol to support efficient acknowledgment operations.

**Transmission of pending data** To transmit the pending list's head chunk and thus transition it to the in-transit list segment, the payload data of this chunk must be provided to the transport protocol implementation. This chunk's data can be accessed by dereferencing the maintained pending list head pointer. In addition to the chunk's payload data, the linked fragmented buffer must further provide all information along with the payload data that it ultimately requires to be sent back as part of acknowledgments (e.g., the address of the current chunk being read back). Furthermore, the linked fragmented buffer implementation should keep track of the address of the previous transmitted chunk, as this address must be included in every segment's transmission as well, for the reasons presented in Section 5.1.3.1.

**In-transit head peeking** The linked fragmented buffer must further support a mechanism called *head peeking*: if the buffer contains chunks in its pending and in-transit list, specific metadata of the head chunks must be made accessible to the transport protocol. This is required to allow ARQ mechanisms to estimate whether retransmission of a chunk is required. An ARQ mechanism could, for instance, use the insertion timestamp of a chunk to estimate whether retransmission is required.

This mechanism can be implemented by monitoring the in-transit head pointer and reading the respective chunk's header if this pointer changes. Providing metadata of the in-transit head chunk is vital, as the *insertion* and *transmission* operations guarantee data to be stored chronologically in the in-transit list, with respect to both the insertion and transmission time. Thus, the ARQ mechanism is supplied

with information about the chunk awaiting acknowledgment for the longest period of time.

**Processing of acknowledgments** In response to received segments, receivers will send acknowledgments covering one or more consecutive segments. Based on these acknowledgments, the transmitter can reclaim memory occupied by acknowledged data and decide whether other data should be retransmitted.

Specifically, in accordance with the findings of Section 2.2.1 and Section 2.2.2, this section shall describe a particular strategy to handle possibly non-consecutive selective acknowledgments in the linked fragmented buffer data structure. Because the transmitter fundamentally transmits all incoming data in-order, except for retransmissions, a selective acknowledgment not acknowledging the transceiver's head in-transit element indicates that some form of loss or reordering must have occurred on the round-trip path from transmitter to receiver. With the presumption of packet reordering being less prevalent in networks utilized by this thesis target application domains (low-complexity networks without dynamic routing and limited parallelism in switching of packets of a single flow), for the intents and purposes of this data structure, it may be assumed that any non-consecutive acknowledgment indicates packet loss. Nonetheless, in accordance with the analysis of Section 2.2.1, it is important for any reliable transport protocol to be tolerant to packet reordering; while treating any reordering event as packet loss may lead to substantial performance impacts, it fundamentally still guarantees reliable delivery of every data segment at the receiver eventually.

Based on this assumption, a strategy to handle acknowledgments for the linked fragmented buffer can be defined; the following text refers to Figure 5.4 for a visual representation of how acknowledgments are processed with respect to the linked list data structures in memory: an acknowledgment must contain pointers to the leftmost (oldest, `E`) and rightmost (newest, `G`) chunk to be acknowledged, as well as the leftmost chunk's predecessor `D`. Furthermore, the acknowledged chunks must be entirely contained in the in-transit list. If the acknowledgment is consecutive, meaning that the leftmost chunk corresponds to the current in-transit list head chunk `C`, the rightmost chunk's header must be read to extract its next chunk pointer, which is then used as the new in-transit head chunk pointer (①). This concludes handling consecutive acknowledgments. If the acknowledgment is instead non-consecutive, this implies that any chunk of the in-transit list before the acknowledgment's leftmost chunk should be assumed to be lost and thus must be transferred to the retransmit list. Thus, the following operations are required: to append one or more chunks of the in-transit list to the retransmit list, the retransmit list tail chunk's next chunk pointer must be set to the current in-transit head chunk address (②). Furthermore, the retransmit list's tail pointer must be set to the address of the acknowledgment's leftmost chunk's predecessor address, supplied as part of the acknowledgment (③). As shown in Figure 5.3, an invalidation of the leftmost chunk's predecessor's next chunk pointer (`D`'s next chunk pointer), pointing to the acknowledgment's leftmost chunk `E`, is not required, given the retransmit list bounds are enforced through the head and tail control data registers (enforcing the list bounds `A` to `D`) instead. With

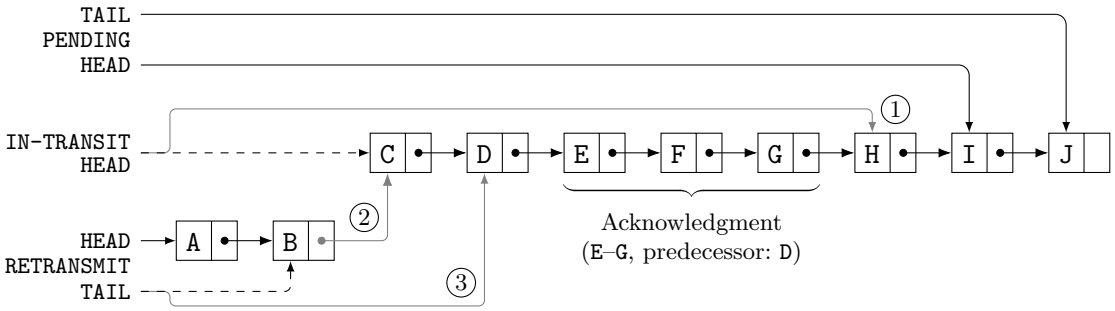



Figure 5.4.: Illustration of steps required to process an acknowledgment. In this particular example, chunks `E` to `F` are acknowledged, to which the acknowledgment provides pointers, in addition to a pointer of the leftmost acknowledged chunk's predecessor `D`. Dashed lines indicate pointer relations to be removed, gray lines indicate new pointer values after the acknowledgment.

the chunks preceding the acknowledgment region attached to the retransmit list, the remaining step is to remove the acknowledged chunks from the in-transit list in an identical fashion as outlined for consecutive acknowledgments above (①).

The strategy presented above does not solve two major remaining issues: while it manages to modify the structures of both the in-transit and retransmit lists to properly represent the acknowledgment and retain internal consistency and boundedness of both lists, because linked lists are not consecutive in memory, these mechanisms do not provide sufficient data to update the count of elements in the in-transit, pending and retransmit lists respectively. Furthermore, this strategy does not invalidate any acknowledged chunks, required to reclaim memory occupied by them. A solution for these issues will be discussed in Section 5.1.3.4. Nonetheless, this strategy enables processing acknowledgments over an arbitrary amount of data (chunks) with a constant number of operations, satisfying the requirements imposed on this data structure.

**Retransmission of data** To account for packet loss on the transmission channel, the transmitter must retransmit data, either in response to messages indicating that data has likely been lost in transit (i.e., non-consecutive selective acknowledgments), or because the employed ARQ mechanism has determined that data has likely not arrived at the receiver.

Given the strategy to interpret and process acknowledgments as presented above, data which is presumed to be lost based on information communicated through acknowledgments is moved from the in-transit to the retransmit list. Thus, locating data scheduled to be retransmit via this mechanism is trivial: if the retransmit list contains at least one chunk, the first to-be retransmit chunk will be located at the address stored in the retransmit head pointer control data register.

When instead the employed ARQ mechanism determines that retransmission of a given chunk is required, based on information provided through the *head peeking*

mechanism introduced above, the in-transit head chunk needs to be read back. Analog to the case above, the in-transit head pointer contains the address of the chunk to read from memory.

In either case, the chunk having been read back should be removed from the respective list. This is because each chunk in the buffer is corresponding to one pending, in-transit or presumed-lost transmission. While a retransmission might itself be lost and require yet another retransmission, this should for all intents and purposes be treated as yet another new transmission. Thus, any transmission consists of a read operation, as well as an insert operation. Removing the read chunk from the respective list involves setting the list's head pointer to the next chunk address stored as part of the chunk's header and decrementing the count of chunks stored in that list.

#### 5.1.3.3. Ephemeral IDs

For the transport protocol to guarantee reliable transmission of all data from the sender to the receiver, and for the sender to validate that a consecutive stream of data has been received without any gaps, every segment of the transport protocol can be tagged with a monotonically increasing identification number (ID). Such IDs thus serve a similar purpose compared to TCP sequence numbers. Because of the fundamental assumption that transport protocol segments correspond to chunks maintained in the linked fragmented buffer data structure, this ID can be stored in the chunk metadata section.

However, the linked fragmented buffer memory model itself does not require such an ID, as outlined through the operations described in the previous section. This is because this data structure is fundamentally agnostic over how the transport protocol handles retransmissions and reassembly of data, it merely provides mechanisms to retain and track single transmissions and handle acknowledgments or ARQ-issued retransmissions accordigly. To the linked fragmented buffer, a retransmission of a given segment is indistinguishable from the first transmission of a segment. Thus, a given segment's transport protocol assigned ID can be treated as an opaque metadata attribute by the data structure.

Nonetheless, the linked fragmented buffer as presented up to this point still has two major deficiencies as outlined under Section 5.1.3.2: first, it cannot, in response to an acknowledgment, efficiently determine how many chunks the acknowledgment references, as well as how many chunks in front of the acknowledged list segment will be transferred from the in-transit to the retransmit list. These values are required to update the respective counters, which are used within various internal mechanisms of the data structure and provided to the transport protocol, for instance to influence decisions about automatic retransmissions or transmission rate limiting. Second, reading back chunks from the retransmit or in-transit list and acknowledging chunks shall remove the respective chunks from the linked fragmented buffer memory. However, no efficient mechanism for reclaiming the memory occupied by this data is defined yet. Specifically, a primitive mechanism to invalidate chunks by writing into memory for each acknowledged chunk shall be avoided, to

maintain the constant time acknowledgment processing characteristic of the data structure as documented above.

In an effort to solve both of these remaining issues, the concept of *ephemeral IDs* is introduced: where a transport protocol may utilize IDs as an alternative to stream offsets such as TCP sequence numbers, fundamentally used to refer to the transferred payload data stream, ephemeral IDs are in reference to individual transmissions or retransmissions of data segments. Thus, a retransmission of a given data segment retains the same transport protocol segment ID as the original transmission, but corresponds to a different ephemeral ID. As consequence, in reference to IP packet reordering as dicussed in Section 2.2.1, a packet containing a retransmission of a given segment will not be identical to the segment's original transmission; this enables retransmissions to be distinguished from out-of-order arrival through packet reordering. While segment IDs are provided upon chunk insertion to the linked fragmented buffer by the transport protocol implementation, ephemeral IDs are generated by the linked fragmented buffer implementation for every inserted chunk in a monotonically increasing fashion.

Such ephemeral IDs can, for instance, be utilized to determine the number of elements between any two chunks in the in-transit list. This is possible because of three basic invariants enforced by the individual operations described above: (i) ephemeral IDs are assigned in a monotonically increasing fashion to every inserted chunk which is appended to the in-transit list, (ii) reading back a chunk in response to an ARQ operation removes the in-transit list's tail element, and (iii) both consecutive and non-consecutive acknowledgments remove all acknowledged chunks and all chunks left of the acknowledgment from the in-transit list. Thus, all chunks contained in the in-transit list are guaranteed to always compose a fully consecutive, monotonically increasing list of ephemeral IDs. Based on this property, the number of chunks $n$ between and including any two chunks $A$ and $B$ contained in the in-transit list can be computed as $n = |\text{ephid}(B) - \text{ephid}(A)| + 1$. Hence, in response to an acknowledgment, the linked fragmented buffer is able to determine the ephemeral IDs of the in-transit list head $H$, the acknowledgment's leftmost chunk $X$ and rightmost chunk $Y$. Based on this information, the number of elements added to the retransmit list $n$ can be computed as $n = \text{ephid}(X) - \text{ephid}(H)$. Furthermore, the number of elements removed from the in-transit list $m$ is computed as $m = n + \text{ephid}(Y) - \text{ephid}(X) + 1$. This information is sufficient to maintain an accurate count of chunks contained in both the in-transit and retransmit lists using a constant number of operations.

In addition to determining the number of chunks between any two chunks in the in-transit list, ephemeral IDs can serve other purposes as well. For instance, a basic requirement for acknowledgments is that all acknowledged chunks must be entirely contained in the in-transit list. But because acknowledgments are to be handled in constant time, and to validate reachability of the acknowledgment's leftmost chunk pointer from the in-transit list head pointer, as well as reachability of the acknowledgment's rightmost chunk pointer from the leftmost chunk pointer would require a search operation in $\mathcal{O}(n)$, this invariant cannot be asserted in practice. However, with the introduction of an ephemeral ID associated with each chunk, this invariant can be validated through operations in $\mathcal{O}(1)$: the linked fragmented buffer can access the acknowledgment-provided pointers to retrieve the ephemeral IDs for the leftmost $L$ and rightmost $R$ acknowledged chunks, as well as the

ephemeral IDs for the in-transit head $H$ and $T$ chunks through the control data maintained pointers. Based on this information, if $\text{ephid}(H) \leq \text{ephid}(L) \leq \text{ephid}(R) \leq \text{ephid}(T)$, all acknowledged chunks must be contained in the in-transit list. This has practical implications for maintaining consistency of the linked list data structure: without any such validations, a duplicate, out-of-order or maliciously crafted acknowledgment could supply arbitrary pointers in memory which are not in reference to chunks contained in the in-transit list and thus incorrectly manipulate the data structure, potentially violating invariants required by subsequent operations and causing data loss and denial of service as a consequence. The ephemeral IDs of chunks can further be attached to segment transmissions and required to be sent back as part of received acknowledgments. This avoids dereferencing pointers $L$ and $R$ received as part of an acknowledgment to recover their ephemeral IDs, an operation which can incur a high latency depending on the type of memory used. Still, the received ephemeral IDs can be used to assert that all chunks of the acknowledgment are contained in the in-transit list, providing sufficient protection against inconsistencies due to duplicated or out-of-order acknowledgments. It should be noted that such receiver-provided pointers and ephemeral ID could lead to maliciously crafted acknowledgments causing inconsistencies in the list structure; however, a detailed security analysis is outside of the scope of this thesis.

A further advantage of the ephemeral ID concept as introduced above is the ability to invalidate chunks in memory, without performing any write operations. Because the linked fragmented buffer implementation maintains pointers to the in-transit and retransmit list head and tail chunks respectively, and can thus determine the ephemeral IDs associated with these chunks, it is able to determine whether a given chunk in memory may still be referenced by either of the two lists: on the one hand, all chunks in the in-transit list are guaranteed to compose a gapless monotonically increasing series of ephemeral IDs, hence the linked fragmented buffer can determine with certainty whether a chunk with a given ephemeral ID must be contained in this list. On the other hand, the sequence of ephemeral IDs composed by chunks of the retransmit list is strictly increasing, but not necessarily gapless and monotonically increasing. Thus, the head and tail element of the retransmit list provide lower and upper bounds of the ephemeral ID space for elements contained in this list. Nonetheless, if for a chunk $X$, with in-transit list head $I_H$ and tail $I_T$, and retransmit list head $R_H$ and tail $R_T$ holds $(\text{ephid}(X) < \text{ephid}(R_H) \vee \text{ephid}(X) > \text{ephid}(R_T)) \wedge \text{ephid}(X) < \text{ephid}(I_H)$, the chunk is not contained in either list and its memory can be reclaimed. Note that $\text{ephid}(X) \leq \text{ephid}(I_T)$ must always hold, therefore no test is required for the case where $X$ would be outside of the bounds of the in-transit list by having an ephemeral ID higher than that of the list's tail element. How this mechanism integrates with the memory allocation algorithm of the linked fragmented buffer is covered under Section 5.1.3.4.

#### 5.1.3.4. Memory Fragmentation and Chunk Allocation

With all fundamental operations of the linked fragmented buffer covered, and the introduction of the ephemeral IDs concept to improve the implementation's efficiency and enforce certain invariants, the major remaining issue is that of memory allocation. Given

that a linked list is fundamentally a data structure composed of multiple, non-contiguous allocations in memory, this is not a trival problem to solve.

Memory allocators for general purpose *heap* storage have been subject to extensive research [32; 33]. As defined by Wilson et al., a memory allocator is an algorithm designed to keep track of which parts in memory already contain an allocation and which parts are *free*. Generally, an allocator cannot know in which order applications release certain granted allocations, creating *holes* in memory (referred to as *external* fragmentation). This is in contrast to *internal* fragmentation, occurring when assinged allocations are larger than requested. The two primary goals for a memory allocation scheme are to allocate memory as compact as possible (minimizing both external and internal fragmentation), with as little time cost (complexity) as possible. While many different allocation schemes exists, the design space of memory allocators are further limited by restrictions imposed by the allocator's user (application), in this case being the linked fragmented buffer mechanisms and data structures. For instance, the linked fragmented buffer's inherent list-based data structures require any existing allocations to not be relocated, as this would invalidate pointers to these allocations and by extension destroy the linked list structure. Futhermore, the allocator shall operate on a chunk granularity; it cannot allocate or free partial chunks as each chunk must be a contiguous memory allocation. While these restrictions are shared with the analysis of memory allocators by Wilson et al., the linked fragmented buffer also enables an unusual optimization for allocators: given that each allocation is guaranteed to start with a header of a known layout, the memory allocator may examine the data stored in any given allocation and may utilize this data to draw conclusions about whether a given memory location is currently occupied.

Generally, allocators over memory utilize some data structure to keep track of existing allocations, and to help find new sufficiently large memory regions. These data structures may either be stored within the managed memory region itself (for instance with *free list* or *indexing tree* allocators) or in a separate region (such as with *bitmapped* allocators; compare [32, pp. 42–61]). Another observation is that many allocators analyzed by Wilson et al. are optimized to allocate regions approximate to some fixed sizes, which allows the allocation mechanism to avoid expensive searches through memory through general-purpose allocation mechanisms and use efficient lookaside data structures of available memory regions fitting the desired allocation [32, p. 41]. Such practices are interesting, because of the fundamental assumption that chunks of the linked fragmented buffer are corresponding to transport protocol segments, and such segments typically have an upper bound on their size: for example, because segments need to be transferred over the underlying packet oriented network layer, it makes sense to derive the maximum transport protocol segment size from (a multiple of) the maximum IP packet size.

In general, the employed allocator for chunks maintained within the linked fragmented buffer can have a significant performance impact, as a chunk allocation is required for every insertion operation. At the same time, the core ideas and mechanisms of the linked fragmented buffer are independent of the allocation scheme used, as long as the invariants documented above are upheld. Thus this section shall present a simple allocation scheme, inspired by concepts of existing allocators as analyzed by Wilson et al., and optimized for specific operations. Further statistical and empirical analyses are required to choose an

allocation scheme best fit for the developed data structure, taking real-world measurements into account.

The allocation scheme developed for this thesis works as follows: all chunk allocations stored in the linked fragmented buffer are of a fixed, maximum size of $n_c = 2^{m_c}$ words, $m_c \in \mathbb{N}_0$, including the chunk header. Each chunk header starts with a 1 bit `valid` field indicating whether it is a valid allocation, and further contains the chunk's ephemeral ID, the payload data length (excluding the header), and the pointer to the next chunk, along with other metadata. By enforcing chunks to be of a fixed size ($n_c$ words), the set of addresses uniquely identifying any chunk in a memory region $M$ of $n_m = 2^{m_m}$ words, $m_m \in \mathbb{N}_0$ and $n_m \geq n_c$, is $C = \{i \in \mathbb{N}_0 \mid 0 \leq i < \frac{n_m}{n_c}\}$ with $|C| = \frac{n_m}{n_c}$. The `valid` field can be used to further define the set of potentially allocated chunks $C_a = \{c \in C \mid M[c \cdot n_c] = 1\}$ and thus the set of guaranteed free chunks $C_f = C \setminus C_a$. Given the observation of the previous section that the bounds defined through the in-transit and retransmit list head and tail ephemeral IDs can be used to further invalidate some chunks in memory which still have their `valid` flag asserted, the set of allocated chunks can be reduced by $C_e = \{c \in C_a \mid \neg\,(\text{ephid}(R_H) \leq \text{ephid}(M[c \cdot n_c]) \leq \text{ephid}(R_T)) \wedge \text{ephid}(M[c \cdot n_c]) < \text{ephid}(I_H)\}$ to the complement $C_{ae} = C_a \setminus C_e$, and the set of free chunks be extended as $C_{fe} = C_f \cup C_e$.

In practice, this means that the linked fragmented buffer has a drastically reduced number of locations where a chunk could start in memory, compared to the number of words in memory ($|C| \ll n_m$). Each of these start addresses contains a marker whether this chunk is currently allocated, and is further annotated by metadata (ephemeral ID) which, in some cases, can deem a given location in memory invalid without deasserting the `valid` marker through a write operation. The allocator's goal is to allocate a chunk $c$ out of the set of available memory locations $C_{fe}$ as quickly as possible. Without any additional data structure keeping track of the addresses contained in $C_{fe}$, the most efficient method to search for a $c \in C_{fe}$ is to search for the first $c_{\text{test}} \in C$ for which $c_{\text{test}} \in C_{fe}$ is satisfied. Such a linear search has a worst-case complexity in $\mathcal{O}(n)$, but is generally more efficient if $|C_{fe}|$ is maximized; in fact, if $|C_{fe}| = |C|$, every $c_{\text{test}}$ satisfies the aforementioned condition. Given that $|C_{fe}| = |C| - |C_{ae}|$, this implies that the set of allocated and non-reclaimable chunks $C_{ae}$ should contain as few elements as possible, with a lower bound set by the set of chunks required to be maintained by the transport protocol $C_r$ composed of chunks in the in-transit and retransmit list.

To implement this scheme, the following operations are required: to initialize the linked fragmented buffer data structure, for every address $c \in C$, the first bit of this word contains the `valid` flag and thus must be deasserted. When a chunk shall be inserted, starting from the last chunk's address or 0, the implementation searches for the next address $c_{\text{test}} \in C$ for which either of the two conditions hold:

- the valid-flag is deasserted, or
- the chunk's ephemeral ID is not contained within the bounds of the retransmit list's head and tail ephemeral IDs and it is lower than the in-transit list's head ephemeral ID.

If such a chunk is found, it can be allocated by asserting its valid marker assigning the next available ephemeral ID (current in-transit tail's ephemeral ID + 1) to it. The inflight list's tail pointer should be set to its address when the memory has been filled with data. The singly linked list structure must be retained, either by updating the next chunk pointer header value of the previous inflight list's tail, or by searching for such a free memory location before writing the predecessor's header (look-ahead allocation).

When chunks are appended to the retransmit list in response to an acknowledgment as described under Section 5.1.3.2, and the retransmit list already contains at least one chunk, this append operation may span the list's head and tail ephemeral IDs over some chunks which are still marked as valid in memory, but are not contained in the in-transit or retransmit list. Thus, the linked fragmented buffer implementation may optionally deassert the `valid` marker of chunks starting at the successor of the previous retransmit tail element, until arriving at the first free chunk as defined above. With this operation, $C_{ae}$ is guaranteed to be equal to $C_r$ and thus the data structure only retains data required by the transport protocol. Without this operation, the data structure might retain allocations for more data than required ($C_{ae} \supseteq C_r$), but eventually these chunks are freed automatically by removing elements from the retransmit list and thus increasing the retransmit list's head ephemeral ID.

This allocation algorithm presented here is fundamentally based on a linear search over memory to find an appropriate allocation, thus it is an algorithm in $\mathcal{O}(n)$. However, through integration with other concepts of the linked fragmented buffer, it enables efficient *reclaiming* of allocated memory in the average case, without requiring memory accesses to invalidate old data in many cases. This allocation scheme only supports allocating fixed-size chunks. In cases of low load on the system, to ensure timely delivery of data, a chunk's payload data may only partially fill such fixed-size allocations, leading to internal fragmentation. However, when the system is under high load, generally the linked fragmented buffer should be able to collect sufficient payload data to fill a given chunk allocation before transmission of this chunk or segment respectively, reducing internal fragmentation. This observation can be considered analog to the small-packet problem identified by Nagle in RFC 896 [34]; Nagle's algorithm will be discussed in the context of the developed protocol within Section 5.4. Furthermore, the allocator may cause subsequent elements to be non-cosecutive in memory (as indicated by *fragmented* in the name of this data structure). However, on a random access memory such as DRAM, the linked-list next chunk pointers enable accessing a given chunk's successor in $\mathcal{O}(1)$.

Finally, in accordance with the data structure, its operations, and the memory allocation scheme established, Figure 5.5 shows the in-memory layout of chunks stored in the linked fragmented buffer.

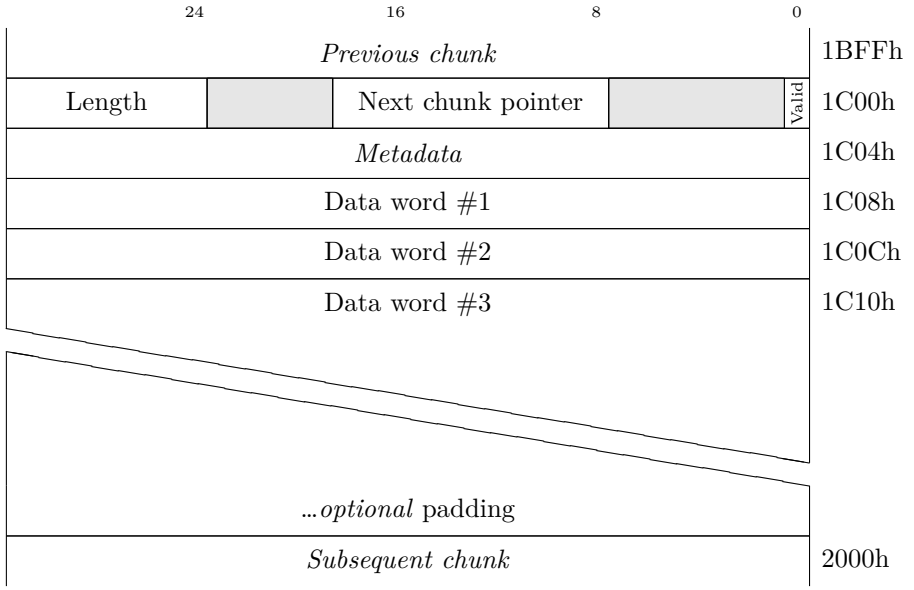



Figure 5.5.: In-memory layout of chunks maintained through the linked fragmented buffer structure. Each chunk contains a header and payload section. The header is a compact (bit-packed) representation of required fields and additional metadata. To efficiently rewrite the *next chunk pointer* field, it imposes certain alignment consraints on its start offset and the start offset of the subsequent field. $n$ bit data words are aligned to $n$ bit.

## 5.2. Flow Control and Congestion Control

With an appropriate memory management architecture defined, supporting all required fundamental operations of a reliable transport protocol such as TCP, and efficiently representable within FPGA systems, it is important to further design the surrounding transport protocol. While the presented memory model certainly is a key component of the overall transport protocol design and a major distiguishing characteristic from existing transport protocols, other mechanisms such as *flow control* and *congestion control*, as well as ARQ algorithms and the receiver implementation may be of equal importance to the protocol's overall performance. This section discusses to what degree flow control and congestion control mechanisms, such as implemented in TCP, are required in the context of this thesis' system model, and how they can be integrated with the linked fragmented buffer design.

As discussed in Section 2.3.2, TCP flow control is a mechanism to avoid overwhelming a slow receiver by a fast sender. Such a mechanism makes sense in the TCP model: fundamentally, it is concerned with providing reliable and in-order delivery of a byte stream over a packet-oriented network, with no assumptions made of the participating host's architecture, processing capabilities and available memory. Because TCP is a protocol designed to provide a communication channel independent of the data transported over this channel, it cannot make any assumptions regarding the involved host's relative available compute and memory resources. This is in stark contrast to the assumptions of this thesis: con-

sidering the system model of Chapter 3 and problem statement of Chapter 4, it can be assumed that the transmitting device is an embedded and specialized measurement device (e.g., a time-to-digital converter). It is further known that the transmitting device will facilitate a hardware implementation of the designed protocol, which should be capable of transmission rates approximating the available link bandwidth, but has only limited memory resources available to cache pending transmission and retransmission data. Furthermore, the data source internal to this device, such as the TDC FPGA core, cannot be reasonably rate-limited. When consolidating all of these prepositions with the fundamental requirement of guaranteeing lossless transmission with eventual in-order arrival of this data, it becomes apparent that this scenario cannot account for insufficient buffering resources at the receiver. This argument is in line with eliminating sources of head-of-line blocking from the developed protocol: given that the developed linked fragmented buffer data structure is no longer subject to head-of-line blocking because of fundamental constraints within the data structure, such as with non-sparse circular buffers, introducing a flow control mechansim as employed by TCP would still effectively introduce a source of head-of-line blocking and thus rescind these advantages. Thus, for the purposes of this thesis, the receiving host must be assumed to have sufficient memory available to buffer any incoming and potentially out-of-order segments until they can be processed further. Under different circumstances, specifically if the data source at the transmitter were to provide mechanisms to limit its output data rate, it may make sense to consider flow control mechanisms governed by the receiver.

While flow control implemented through receiver feedback does not appear practical given this thesis' presumptions, some form of congestion control in response to congestion indicators in the network is required. Whereas the receiver must be able to buffer any out-of-order transmissions and can be dimensioned appropriately, the intermediate network is more unpredictable. Even in small, controlled laboratory networks utilizing few ($< 10$) intermediate network stations, the available end-to-end bandwidth heavily depends on the link speeds, employed network structure, switch or router architectures, and can vary based on route changes or simultaneous transmissions of other network participants. Many of these factors can be out of the control of the network user. Naturally, if the available network bandwidth is insufficient to transmit incoming measurement data over prolonged periods of time, and the transmitter is unable to buffer more incoming data, data loss can be unavoidable.

The fundamental purpose of congestion control is, as the name suggests, to avoid congestion at the bottleneck network link(s), which would result in excessive loss events for this and other network flows. This implies that, to take full advantage of the end-to-end path's bandwidth, any such algorithm must either know the exact share of bandwidth it may utilize on a given path, or receive feedback from which it can approximately derive its allowed share of bandwidth. TCP's congestion control mechanisms, as described in Section 2.3.3, form a type of loss-based congestion control. They utilize losses as implicit feedback of the network indicating congestion on some link(s) [17, p. 572]. Based on this information, TCP maintains a window keeping track of the amount of data it may inject into the network at any given time, without causing excessive loss events as a consequence of overwhelming the bottleneck link. As outlined in Section 2.3.3 and illustrated in Figure 2.2, TCP utilizes slow start and congestion avoidance mechanisms to, in response to

loss events, recover effective link utilization in logarithmic time and continue to probe link bandwidth limitations linearly.

However, such loss-based congestion control schemes suffer from an inherent issue: given they are in response to loss events, they only limit the utilized bandwidth when congestion already occured on a path. Using base TCP of RFC 793 as an example, it would first have to recover this loss event before additional data could be transferred. Even if selective retransmissions are supported, reverting back to the slow start congestion control algorithm severely reduces overall utilized bandwidth. In case the congestion event has only caused losses of a few select number of segments, and selective acknowledgments are employed, fast retransmit and fast recovery can help maintain a sustained effective utilization of link bandwidth.

Yet, there is another indication of congestion available to the transmitter, which is not in response to loss events: the round trip time (RTT) from transmitting a given segment to receiving its acknowledgment. Assuming a fixed path over some network links, where each link has a constant propagation time, the propagation delay of packets over this path can only change because of intermediate network stations. If the behavior of these intermediate network stations with respect to their imposed delay on packet transmissions is known, an end-to-end measurement of the packet propagation delay can provide limited and cumulative information about the state of intermediate network stations. As explained by Tanenbaum and Wetherall, intermediate stations in packet-switched networks such as IP generally utilize a store-and-forward scheme to pass data from one link to another. In contrast to circuit-switched networks, packet-switched links do not usually assign fixed (time, frequency, etc.) slots for transmissions of a given circuit, but rather packets are placed on a given link one at a time following a set of rules. If multiple packets destined for a single destination link arrive at a packet switching node simultaneously, these packets will be held in memory at this node and sent over the target link eventually, according to some scheduling discipline [17, p. 356]. Such a scheduling discipline should avoid reordering packets of a single flow (as defined in Section 2.2.1), as this would manifest as packet reordering at the receiver. It is this property of packets being delayed due to buffering in case of the link being busy, along with a FIFO queueing discipline for packets part of a single flow, which enables using the overall propagation delay as an indicator for the amount of queued packets on the end-to-end link. Given sufficiently large (*deep*) queues in the intermediate network elements, transport protocols can utilize this metric to predict congestion loss before it happens. Figure 5.6 shows a measurement of the observed RTT (blue) and packet loss (red) as a function of the link utilization, performed within a deterministic ns-3 simulation. The simulated topology contains two hosts, each connected to a router with a 1 Gbit/s link. The two routers are connected with a 50 Mbit/s link. As soon as this bottleneck link bandwidth is exceeded, packets are buffered within a router, increasing the observed latency. One the packet buffers are full, packet loss occurs. Thus, for any congestion control scheme to avoid such loss events while converging to the available link bandwidth, increases in the transmission rate should be approximately inversely proportional to the observed end-to-end propagation delay.

Transport Informed by MEasurement of LatencY (TIMELY), as described by Mittal et al., is such a congestion control mechanism utilizing measurements of the observed RTT

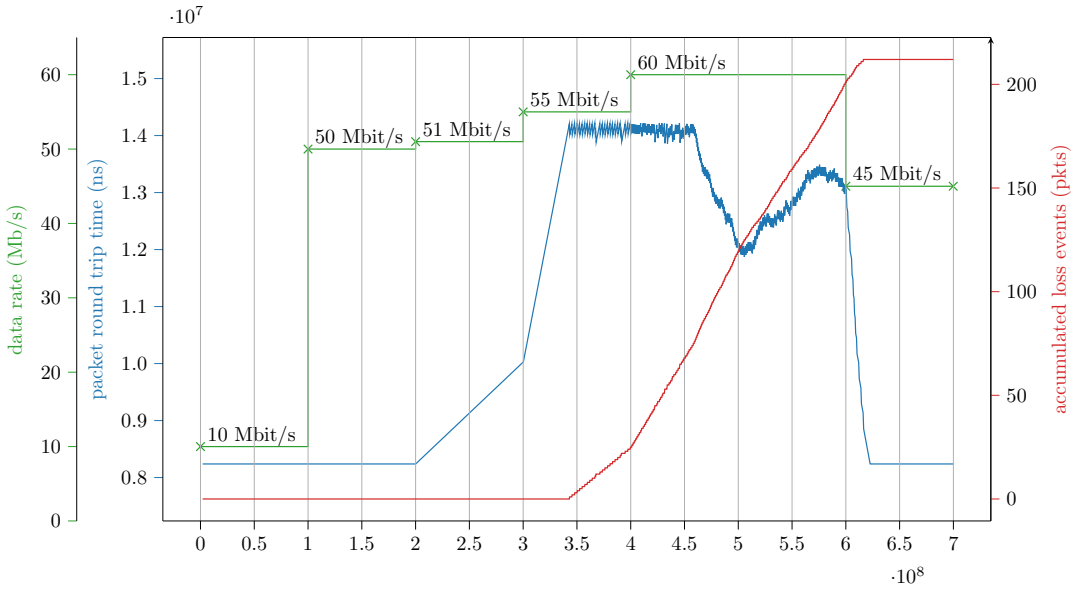



Figure 5.6.: Observed network path RTT and loss events as a function of link utilization. Data obtained using a deterministic ns-3 simulation with two hosts connected to two routers, in turn connected with a 50 Mbit/s bottleneck link. $x$-axis shows simulation time in nanoseconds.

to estimate network congestion [35]. By controlling the transmission rate based on the observed propagation delay, it is a *rate control* approach to congestion control. TIMELY is especially interesting in the context of this work, as it deliberately does not target only TCP, but reliable transport protocols in general. Furthermore, TIMELY makes a basic assumption easily met in the context of this thesis' system architecture: to acquire a very precise estimate of the RTT, which is especially important for low-latency (microseconds to tens of milliseconds) connections found in datacenter networks or laboratory networks (as targeted within this thesis), this mechanism mandates packet transmission and reception timestamps to be acquired within the NIC itself. Given that this thesis is targeting FPGA platforms implementing the Ethernet MAC in reprogrammable logic, such timestamping logic can be supplemented. In fact, for the LiteEth Ethernet MAC, the author of this thesis has already integrated this functionality. TIMELY also mandates that the receiver is either able to timestamp incoming and outgoing packets and supply information about the time between reception of a segment and transmission of its acknowledgment, or the NIC can issue acknowledgments for segments in hardware. The latter, while possible for TCP, is highly unlikely for a custom network protocol. Fortunately, many NICs feature packet timestamping capabilities for arbitrary packets already, in an effort to support the Precision Time Protocol (PTP) of IEEE 1588. If this feature is not supported, timestamps have to be acquired in software instead, with a corresponding decrease in accuracy [35, pp. 537–538].

Given the amount of prior research in this field, the applicability of congestion control algorithms to the concepts of HELIX, and especially the compatibility with the developed linked fragmented buffer memory architecture, it appears reasonable to reuse preexisting

congestion control algorithms such as TIMELY. By combining feedback through (missing / non-consecutive) acknowledgments along with an estimation of the path RTT as inputs to the congestion control algorithm, and using the number of chunks in the in-transit list as an indicator of the current network load, TIMELY appears to integrate well with the overall target architecture of this thesis. The output of the congestion control mechanism can then be used to limit the transmission rate of individual network packets. A mechanism to estimate the current RTT is required independent of the congestion control scheme for implementation of an ARQ mechanism; its implementation is further described in Section 6.3.1.

## 5.3. Receiver Model and Architecture

Before the design of the elaborated protocol can be presented in its entirety, it is important to also consider the architecture of a receiver implementation, compliant and cooperating with the presented memory model and congestion control mechanism. Specifically mechanisms for reassembly of incoming packets into an in-order stream of data words, as well as formation and eventual transmission of acknowledgments are important to build a coherent and formally sound protocol.

As required by the linked fragmented buffer design, each segment arriving at the receiver will contain one or more payload words, along with certain metadata. Some of this metadata is mandated by the linked fragmented buffer, whereas other metadata is determined by the surrounding transport protocol. Specifically, as elaborated within Section 5.1.3.2, the linked fragmented buffer requires each segment to be annotated with the ephemeral ID and address of the transmitter's copy of this data in memory (the corresponding chunk), as well as the preceding chunk's address in memory. Nonetheless, this metadata is insufficient for the receiver to reassemble the received data into a gapless, in-order stream of data words. This is because, as discussed previously, the ephemeral IDs refer to individual transmissions of data; hence retransmissions of a given segment are annotated with an ephemeral ID different from the initial transmission. Thus, the transport protocol must further provide a strictly monotonically increasing ID for segments, persistent across retransmissions.

The receiver must utilize these data to both provide the gapless, in-order data stream as required, and form acknowledgments respectively. We shall consider these issues separately, focusing on generation of acknowledgments first. In theory, acknowleding incoming data is a trivial operation: it is sufficient to simply acknowledge one received segment at a time, using only metadata provided along the received segment. However, this has two major disadvantages: first and foremost, while the linked fragmented buffer data structure is able to process acknowledgments in constant time, each acknowledgment still incurs transformations of the linked list structures maintained at the transmitter, with a bounded number of memory operations, and as such still has a performance impact. Furthermore, generating an acknowledgment message for every received segment imposes additional load on the receiver and network, and can thus potentially decrease the achieved goodput on the

connection [36, pp. 96–97; 37]. For this reason, RFC 1122 introduces and RFC 5681 further specifies *delayed acknowledgments*: through this mechanism acknowledgments, given their cumulative nature, can be delayed for up to 500 ms, allowing to acknowledge a number of consecutive segments through a single message, where for TCP an acknowledgment should be sent at least for every second full-size segment [36; 21]. This mechanism is compatible with the selective acknowledgment structure imposed by the linked fragmented buffer, by being able to specify a range of segments through the leftmost and rightmost segment of that range respectively. Delayed acknowledgments fundamentally delay the time until memory occupied through buffered chunks corresponding to unacknowledged segments at the transmitter can be reclaimed. Furthermore, insertion of additional delays makes RTT-based congestion control algorithms less effective, given their estimate of the network-imposed end-to-end delay may be significantly skewed. Thus, a good balance between system and network load caused by acknowledgments and timely acknowledgment of data is required. Going further, Allman continues in the argument against delayed acknowledgments, finding that this mechanism may even reduce goodput compared to immidate acknowledgments of all segments [37]. Empirical studies are required to determine the best strategy given the developed protocol, as well as the transmitter implementation's performance impact of processing acknowledgment messages.

That said, this thesis proposes a strategy for receiver implementations to generate acknowledgments: the receiver shall maintain a data structure containing the following information:

- `valid`: whether the data contained in this data structure is valid
- `left_pred_addr`: the pending acknowledgment's predecessor chunk address
- `left_addr`: the pending acknowledgment's left chunk address
- `left_ephid`: the pending acknowledgment's left chunk ephemeral ID
- `right_addr`: the pending acknowledgment's right chunk address
- `right_ephid`: the pending acknowledgment's right chunk ephemeral ID
- `rx_time`: the time of reception of the left chunk's segment

Upon reception of a segment, the receiver shall adhere to the following strategy:

1. If `valid` is asserted, go to 2, otherwise continue with 4.
2. If the received segment's ephemeral ID is equal to `right_ephid` + 1 (meaning that the current pending acknowledgment can be extended to include this segment), go to 3, else continue at 5.
3. Set `right_ephid` to the ephemeral ID of the received segment. Set `right_addr` to the address of the received segment. This concludes handling of the received segment.
4. Store the received segment's ephemeral ID in `left_ephid` and `right_ephid`. Store the received segment's address in `left_addr` and `right_addr`. Store the reception timestamp in `rx_time` and set a pending acknowledgment timer to invoke step 5 after `rx_time` + $\delta$, where $\delta$ is the maximum delay acknowledgments may experience at the receiver. Assert `valid`.
5. Form an acknowledgment from the pending data and send it. Deassert `valid`. Disable any pending acknowledgment timer. If this is in response to receiving a segment and not in response to the acknowledgment timer expiring, go to 4.

This strategy aggregates selective acknowledgments for consecutive segments over a time window $\delta$, which can then be efficiently processed in constant time at the transmitter, reducing the acknowledgment-induced overhead. However, out-of-order delivery of segments, potentially indicating loss events in the network, cause the receiver to immediately issue a selective acknowledgment. As such, this mechanism provides a reasonable balance between minimizing acknowledgment overhead at the transmitter, receiver, and on the network, while reparing loss conditions in a busy stream quickly. Only when loss events occur on a stream which are not immediately followed with subsequent transmitted packets will ARQ-issued transmissions be delayed. However, if loss events are not immediately followed by subsequent packets, this indicates that the transmitter is not particularly busy, and thus it should have sufficient buffer capacity to handle these cases.

Lastly, the mechanism to reassemble an in-order, gapless data stream at the receiver is to be described. While a conceptually simple mechanism, this thesis shall propose appropriate data structures to allow for efficient stream reassembly at the receiver: to avoid reallocation and relocation of payload data while processing acknowledgments, received segments should be stored in volatile memory once upon reception, with all other data structures operating on pointers to locations in memory. This is not unlike Linux's `sk_buff` data structures as documented in Section 5.1.1. To ultimately reconstruct an in-order data stream, the receiver should insert received segment's payload data pointers into a FIFO queue, if and only if the segment to be inserted is consecutive to the previous inserted segment. For any out-of-order segments received, this thesis proposes to insert them into a hash table, indexed by their transport protocol ID. Furthermore, the implementation should retain the value of the next expected in-sequence ID. Then, the mechanism to handle incoming segments works as follows: check whether the received segment's transport protocol ID matches the next expected ID. If it does, insert it into the in-order queue of received segments. Furthermore, increment the ID by 1 and try to remove the element with this index from the hash table, inserting it into the in-order queue. Repeat this process until the hash table does not contain an element with the given ID.

## 5.4. The HELIX Protocol

Finally, based on the developments of the previous sections, the resulting HELIX protocol can be described in its entirety. This section will present the developed network protocol structure, including its on-wire format, reiterate the required message exchanges, and present other details not yet discussed.

HELIX is largely agnostic over its underlying network protocol. While this thesis has performed its analyses largely based on the combination of Ethernet and IP, every protocol providing similar characteristics can potentially be used to transfer HELIX messages. The most interesting alternative to using IP directly is to use UDP over IP as a carrier for exchanged messages. While UDP features additional checksums for data integrity, it does not generally supplement any reliability guarantees. However, it features a 16 bit service identifier (*port number*), such as the one TCP provides. This enables differentiating

individual HELIX connections between the same pair of hosts, and has native support within virtually every general-purpose operating system.

The two basic types of HELIX messages exchanged between hosts are data transmissions and acknowledgments. In contrast to TCP, given that HELIX only supports unidirectional payload data flow, it does not combine these two types of messages into a single transmission unit. To be able to distinguish between these message types, every HELIX message should start with a common header. Indicating the wrapped message (*submessage*) type at the start of the packet allows efficient decapsulation with pipelined stream processing architectures, as shown in Section 3.3.1. Furthermore, the header starts with a *magic sequence* of `HLX`, encoded using the American Standard Code for Information Interchange (ASCII) into 3 octets, useful to identify invalid UDP datagrams. Finally, HELIX is by design a parametrizable protocol: implementation constants such as the data word width, chunk size, memory region length and ephemeral ID width are not specified as part of the protocol, but depend on the individual hardware or FPGA implementation of it. Because a serialization format dynamically encapsulating these parameters would be inefficient, the HELIX network protocol on-wire format depends on these constants to be known ahead of time, with the network encoding being determined through a deterministic set of rules, based on these parameters. Given that the encoding and parsing of messages is fundamentally dependent on these parameters, it is important to ensure that the HELIX transmitter and receiver are in sync. For this reason, the common HELIX message header contains a 4 byte parameter digest sequence, which can be used to validate that transmitter and receiver use identical parameters. To retain the usefulness of this value, the message header is, up to and including this sequence, fundamentally not dependent on these parameters. Figure 5.7 shows the structure of this common header. The 4 byte parameter digest is determined by encoding the well-defined set of parameters (as shown in Figure 5.8) using the JavaScript Object Notation Canonicalization Scheme (JCS) as defined in RFC 8785 [38]. The Secure Hash Algorithm 2 with 256 bit output (SHA-256) is used to derive a digest from the resulting encoded string. The first 4 octets of this digest as HELIX's parameter digest value. All fields shall be encoded as big endian (network byte order).

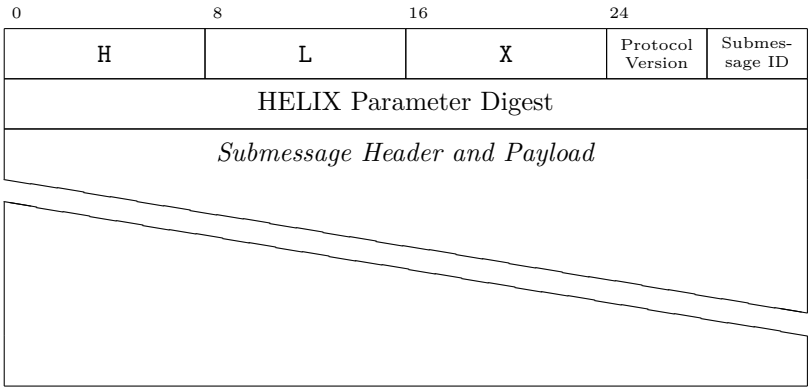



Figure 5.7.: On-wire format of every HELIX message. The common HELIX message header encodes the magic byte sequence `HLX`, the protocol version, submessage type, and the 32 bit parameter digest.

| | |
|---|---|
| `{` | |
| `"_parameters_version": 0,` | Version number incremented on every change to this interchange format. |
| `"max_chunk_len": 8160,` | Maximum payload length of a segment / chunk in memory (in bytes). |
| `"chunk_addr_width": 16,` | Width of the address of a given chunk in memory (in bit). |
| `"chunk_id_width": 48,` | Width of the transport protocol chunk ID field (in bit). |
| `"ephemeral_id_width": 64,` | Width of the linked fragmented buffer ephemeral ID field (in bit). |
| `"chunk_tx_part_field_width": 4,` | If a given segment is to be split over multiple packets of the underlying network protocol, this field specifies the width of the `part` field of transported messages (in bit). If `max_chunk_len` and the HELIX message headers fit within a single packet, this field is set to 0. |
| `"word_length": 32` | Width of a data word (in bit). |
| `}` | |

Figure 5.8.: Example of a HELIX parameter specification shared between the transmitter and receiver implementation. The information contained within this data structure is sufficient to determine the on-wire layout for all HELIX messages. The HELIX parameter digest is determined by encoding this JSON object through JCS, hashing it with SHA-256, taking the first four octects of the resulting hash.

Whereas all fields in the common HELIX message header are aligned to 8 bit, with the 32 bit HELIX parameter digest aligned to 32 bit, such alignment would be inefficient to maintain for the HELIX payload data submessage type. For this submessage, being assigned the *submessage ID* `0`, and succeeding after the common HELIX message header, the header field widths are defined through the individual protocol parameters respectively. Based on the example presented in Figure 5.8, Figure 5.9 shows the payload data submessage header layout derived from these parameters. Given that fields are no longer aligned on a byte boundary, they are to be encoded sequentially from most to least significant bit. However, appropriate padding is inserted to align the subsequent data words, which are required to have a bit-width being a power of two, with at least 8 bit; these data words are aligned on a 8 bit boundary for efficient processing at the receiver. To reduce the relative overhead of chunk headers stored in the linked fragmented buffer, the maximum chunk payload length and HELIX headers may exceed the underlying network protocol's maximum transmission unit (payload data length). For this reason, it is possible to split a single segment, corresponding to a single chunk, into multiple parts. In this case, the *part* field contains a counter incremented for every transferred part of such a segment. The combination of ephemeral ID and *part* field allows the receiver to reassemble split segments. For simplicity the receiver may assume that parts are not reordered in the network; this implies that reordering or corruption of any part of a segment transmission is equivalent to a loss of the entire segment.

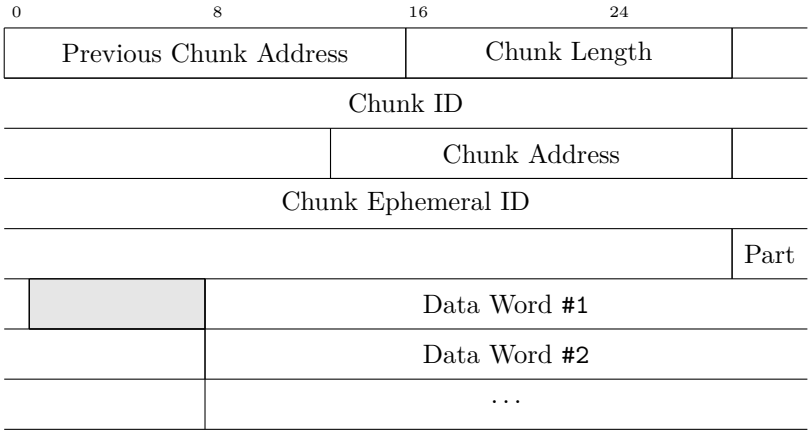



Figure 5.9.: On-wire encoding of a HELIX payload data submessage (submessage ID `0`), following the parameter specification of Figure 5.8.

Analog to the HELIX payload data submessage layout, acknowledgment submessages (ID `1`) contain a compact packed representation of the following fields, in the specified order: the address of the chunk preceding the leftmost chunk of the acknowledgment range, the address of the acknowledgment's leftmost chunk, the ephemeral ID of the leftmost chunk, the address of the rightmost chunk, and the ephemeral ID of the rightmost chunk. The message may be padded with arbitrary data to comply with any alignment constraints imposed by the underlying network protocol.

One aspect which the described design does not cover is that of connection establishment and management. As presented within Section 2.3 and Appendix A, TCP employs an elaborate connection establishment and management scheme to be able to use an unreliable packet-oriented network layer as a basis to establish a reliable connection-oriented communication channel. Considering the target application domains of this thesis, and the complexity involved in an architecture such as the one used by TCP, it makes sense to use established technologies to solve this issue. In particular, the developed protocol is not suitable to establish a bidirectional control connection for the measurement data receiver to communicate with the data acquisition device. Such a connection is assumed to be required in most cases. Given its reduced performance requirements, this connection can be implemented on an external microcontroller, or a processor synthesized within the device's FPGA itself. The software running on this processor can then use established protocols such as TCP to establish a reliable communication channel, over which the required messages for connection establishment and management of HELIX are exchanged. Specifically, it can be used to inform the sender of the receiver's IP address and the port HELIX data stream.

To automatically retransmit lost segments, HELIX has the inherent notion of *lost* segments based on selective acknowledgment feedback. These segments can be transmitted as soon as the congestion control mechanism allows. However, for data still presumed to be in-transit and in cases of a lack of acknowledgments, a different mechanism to retransmit data has to be employed. This problem has already been solved in TCP through the concept of

a retransmission timeout (RTO) [20, p. 41]. This mechanism requires an estimate of the path RTT, which is maintained for the congestion control mechanism anyways.

Finally, the remaining aspect to discuss is Nagle's algorithm and its interaction with the concept of *delayed acknowledgments*. Nagle's algorithm has been designed to avoid significant network overhead caused by the transmission of small packets, where packet headers incur a relatively significant cost compared to the conveyed payload data. Thus, Nagle proposes the following algorithm: if a packet is full, send it. If a packet is not full, send it as long as the receiver has acknowledged all previously sent packets [34]. Whereas HELIX is not primarily concerned with minimizing its network overhead, because each transmitted segment also corresponds to a fixed-size chunk allocation in memory, it makes significantly more effective use of its memory resources when full-size packets are sent. Thus it makes sense to adopt Nagle's algorithm or a slight variation of it. If this algorithm is combined with delayed acknowledgments, as outlined previously, it can cause data to arrive with a significant delay (up to the maximum acknowledgment delay and RTT) if there is a partial segment to transmit [39]. While this poses an issue for interactive systems or at the end of transfers, HELIX is designed for bulk data transfer only. Hence such delay is tolerable in the case of reduced system load.

To conclude the protocol design section, the assumptions with respect to the provided network and the behavior of HELIX shall be reiterated shortly: HELIX fundamentally assumes that reordering of packets in the utilized network is not prevalent. While it is tolerant of seldom reordering events, these events imply a retransmission of at least the affected data; thus reordering significantly reduces the protocol's efficiency and causes data to occupy transmitter memory for longer periods of time. The protocol assumes that the receiver always has sufficient memory available to buffer any segments as fast as they are transmitted, even if there are temporarily gaps in the received data stream. Thus, no flow control mechanism is employed. The protocol features limited resilience to network congestion by integrating a congestion control mechanism. Nonetheless, if the available network bandwidth is insufficient to match the data acquisition rate at the transmitter for prolonged periods of time, data loss may be unavoidable.

# 6. Implementation

In order to validate the elaborated system concepts and the resulting protocol developed in Chapter 5, this thesis shall provide a proof of concept implementation of the described components. Given the problem statement of Chapter 4 and the fundamental presumptions of this thesis, it is of particular importance for the proof of concept implementation to validate that the developed solution is both correct and applicable to hardware platforms (FPGAs) as employed by the target measurement devices. It is most important to provide proof of concept implementations for the novel components and concepts developed as part of this thesis, specifically the linked fragmented buffer structure, to support the claims regarding this data structure's correctness and applicability to the problem at hand as well as performance. Thus, this chapter describes the implementations of individual components developed throughout this thesis to provide the foundation for analyses of correctness and performance in the subsequent chapter.

## 6.1. FPGA Implementation Environment

As described in the system model of Chapter 3, the measurement devices this thesis targets are based on FPGAs. For such an FPGA device to be useful, however, it needs to feature certain external peripherals such as DRAM modules or Ethernet connectors, as well as an internal structure into which the implemented logic components can be integrated. As both of these aspects will be provided by the respective measurement device itself, for the intents and purposes of this thesis, a preexisting FPGA board integrating the required peripherals, as well as a free and open-source FPGA system on a chip (SoC) framework are used.

The FPGA board used for this thesis is the NetFPGA-SUME board [40]. This board features a Xilinx Virtex-7 690T FPGA, connected to various onboard peripherals. Most interesting for this thesis are two Small Outline Dual Inline Memory (SO-DIMM) slots, each holding a 4 Gbyte Double Data Rate 3 Synchronous Dynamic Random-Access Memory (DDR3-SDRAM) module. Furthermore, four 10 Gbit/s Ethernet SFP+ interfaces are connected to the FPGA, allowing it to directly interface with other 10 Gbit/s-capable Ethernet devices via electrical (copper) or optical links.

Furthermore, this thesis uses LiteX as an FPGA logic design framework, providing basic peripherals and a structure into which the custom logic components can be integrated [41]. For instance, LiteX provides a unified memory bus, integrates processor designs (so-called

*softcores*) and contains components implementing basic input/output functionality, such as a universal asynchronous receiver-transmitter (UART) interface. Finally, LiteX can serve as a basic portability layer between different FPGA boards, which reduces friction when adopting the implemented logic designs to different boards or entirely different FPGAs. More important than the core LiteX project, however, is the larger ecosystem built around it, as well the Migen HDL used throughout this ecosystem: Migen implements a HDL as a Python domain-specific language (DSL), by modeling logic circuits through a custom abstract syntax tree (AST) implemented using objects within the Python programming language [42; 41]. This hardware description language is particularly useful in the context of this thesis. Because the developed memory model and the transport protocol are designed to be parametrizable, adapting to various hardware limitations and user-defined constraints, using Python as a DSL to describe hardware logic means that these parameters can be used to perform arbitrary computations and adjust the output accordingly. Furthermore, it can, while building the logic design, automatically output a parameter JSON file as shown in Figure 5.8.

As described in Section 3.3 of the system model, and as a logical consequence from the design established in Chapter 5, the transmitter logic inherently needs to interface with an Ethernet and IP network stack, as well as some DRAM memory controller. Conveniently, *LiteEth* and *LiteDRAM* are projects of the larger LiteX ecosystem, providing sophisticated implementations of such components respectively. While LiteEth contains much of the infrastructure required to interface with remote hosts via 100 Mbit/s and 1 Gbit/s Ethernet links and the IP and UDP protocols, support for 10 Gbit/s Ethernet had to be implemented as part of this thesis and has been integrated back into the LiteEth project codebase. Specifically, the pipelined streaming data path used throughout LiteEth had to be changed from being 8 bit wide, which would require the core to run at a clock frequency of 10 Gbit/s / 8 bit/cycle = 800 ps/cycle = 1.25 GHz, which exceeds the timing constraints of the employed FPGA even for low-complexity logic. Instead, the core is to be run at 64 bit/cycle, allowing to process 10 Gbit/s Ethernet at a clock frequency of 156.25 MHz. This change required substantial rearchitecting of the mechanism to encapsulate and decapsulate packet headers (*Packetizer* and *Depacketizer* components). Furthermore, the XGMII interface to integrate with 10 Gbit/s Ethernet PHYs had to be integrated with the LiteEth MAC, including appropriate inter-frame gap maintenance and integration of the deficit idle count mechanism as defined in IEEE standard 802.3 [4, sec. 4, 46.3.1.4].

Given this environment, the individual components to be validated for this thesis can be implemented as self-contained Migen modules, having access to the required interfaces of the LiteDRAM memory controller to interface with DDR3-SDRAM memory modules, and access to the LiteEth core to integrate with the provided UDP socket hardware interface. This chapter continues by describing the implementation of the linked fragmented buffer and, based on this component, the implementation of the HELIX protocol.

## 6.2. Linked Fragmented Buffer Implementation

The proof-of-concept implementation presented throughout this section is based on a slightly modified version of the linked fragmented buffer design as presented in Section 5.1.3. Specifically, it does not feature a combined in-transit and pending chunk list. Instead, all chunks inserted into the data structure are immediately contained within the in-transit list. Hence implementation of the operation of *transmission of pending data* of Section 5.1.3.2 is not required. This simplified design does, however, mandate an additional and separate buffer to hold incoming measurement data until it can be transmitted over the network and inserted into the linked fragmented buffer, respectively. This simplified architecture is feasible to validate the correctness of the linked fragmented buffer design and to evaluate its performance, as extending it towards the design elaborated in Chapter 5 does not introduce additional manipulations of in-memory data. In fact, the design of Chapter 5 reduces overhead by avoiding copies from the first-stage buffer of incoming measurements into the second-stage linked-fragmented buffer for in-transit data while necessitating slightly more complex management logic. Hence the implemented module will have worse or comparable performance to an implementation fully adhering to the presented design of Chapter 5.

The linked fragmented buffer component will primarily interface with the LiteDRAM memory controller to read from and write to the attached DRAM memory, respectively. In turn, it exposes interfaces implementing the operations described throughout Section 5.1.3.2. It is essential to understand the interfaces provided by LiteDRAM first to devise the logic required to implement the data structure in its entirety.

### 6.2.1. DRAM Memory Interfaces

Through its nature, DRAM memory requires complex controller logic to operate. For instance, memory cells need to be refreshed regularly to retain their contents [43, pp. 76, 88–89]. Furthermore, DRAM memory requires some strict timing constraints to be adhered to by the controller. LiteDRAM is an implementation of such a DRAM controller, abstracting the complex hardware constraints of the DRAM modules themselves and providing conceptually simple interfaces for application logic to interface with the memory. Notably, the provided interface's data width is largely independent of the DRAM module's data width. This is because the DRAM module operates at a different clock frequency than the application logic. Thus, LiteDRAM will automatically determine the appropriate interface width to be able to transfer sufficient data to satisfy the available DRAM interface bandwidth. For instance, for the NetFPGA-SUME board with `MT8KTF51264` DRAM modules operated at 625 MHz, the application logic operating at 156.25 MHz is provided with a 512 bit-wide interface for a theoretical memory bandwidth of 10 GB/s.

To access the DRAM memory and perform required refresh operations efficiently, LiteDRAM features a *look ahead* interface. Before accepting data to write or providing read data, LiteDRAM consumes a limited number of *commands*, indicating the addresses to

operate on and whether a read or write operation is requested. Thus, it is important to enqueue multiple operations, and not to wait for a single operation to be completed before performing the next, in order to utilize the available memory bandwidth efficiently. The controller additionally features embedded functionality to provide multiple such ports to the application logic and automatically process requests from any provided ports.

To reduce the complexity of the provided native LiteDRAM ports, where user logic must maintain commands and write or read data separately, LiteDRAM further provides so-called `DMAWriter` and `DMAReader` modules. The `DMAReader` can enqueue a limited number of read-operations, which will then be processed sequentially and transparently by LiteDRAM, with the results placed into a FIFO buffer for later processing by the application logic. The `DMAWriter` can enqueue a limited number of write-operations, accepting both the memory address and data to be written in a single cycle. It will then transparently process these write operations. While these modules provide simple interfaces to interact with the DRAM controller from application logic, specifically for the `DMAWriter` being buffered means that the time it takes from a write operation being accepted to the write being committed is generally unknown; this problem will be covered further in Section 6.2.3.

For the remainder of this chapter, we can conclude that the interfaces exposed to the user application by the DRAM controller are structured as outlined in Table 6.1. For each of the streaming pipelined interfaces, a bus transaction occurs when the producer asserts `valid` with all associated data, and the consumer asserts `ready` in the same clock cycle.

| Read Port | | Write Port | |
|---|---|---|---|
| $C \rightarrow A$ | `cmd_ready` | $C \rightarrow A$ | `ready` |
| $A \rightarrow C$ | `cmd_valid` | $A \rightarrow C$ | `valid` |
| $A \rightarrow C$ | `cmd_address[n]` | $A \rightarrow C$ | `address[n]` |
| $A \rightarrow C$ | `res_ready` | $A \rightarrow C$ | `data[m]` |
| $C \rightarrow A$ | `res_valid` | | |
| $C \rightarrow A$ | `res_data[m]` | | |

Table 6.1.: DRAM controller interfaces using the `DMAReader` and `DMAWriter` modules for reading and writing memory data, respectively. Each signal is either from DRAM controller $C$ to application logic $A$ or from $A$ to $C$. The memory address is $n$ bit wide, the DRAM controller data word is $m$ bit wide.

### 6.2.2. Linked Fragmented Buffer Implementation Architecture

Based on these DRAM interfaces, an architecture for the implementation of the linked fragmented buffer can be developed. While the design developed throughout Section 5.1.3 fundamentally describes a single, coherent data structure and memory model, it makes sense to divide its implementation into individual components of lower complexity. This is motivated by the fact that in hardware systems such as FPGAs, all logic is executed in parallel. Thus, if two operations were to commence simultaneously, updating a single in-memory location or a control data register such as the in-transit list head pointer may

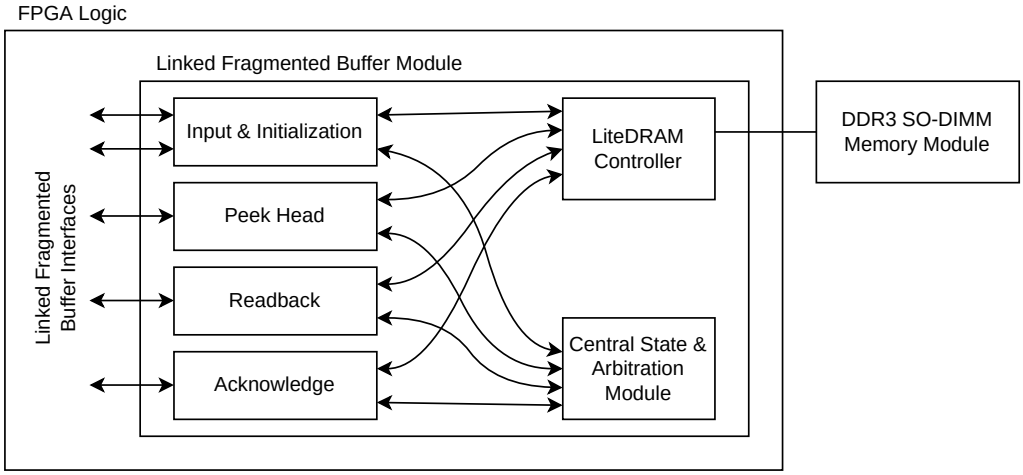



Figure 6.1.: Linked fragmented buffer implementation architecture and intergration with the LiteDRAM memory controller. Each supported operation is implemented through its own independent module, while central state is managed and orchestrated by the central state and arbitration module.

result in an unintended inconsistency in the data structure, not unlike *race conditions* with insufficient synchronization in software systems. By implementing each of the individual operations as described within Section 5.1.3.2 in its own module, and synchronizing accesses to shared data through a central arbitration module, maintaining the data structure's consistency is a matter of appropriate synchronization of each module's requests. A first implementation of this linked fragmented buffer did not define such clear abstractions and was consequently suffering from severe inconsistencies in the data structure, especially when different operations were executed simultaneously. Figure 6.1 shows an illustration of this architecture.

The central arbitration module is responsible for holding the following state in registers (i.e., device internal SRAM), performing all modifications of it, and orchestrating all manipulations of the linked-list structures in memory:

- `it_head`: the in-transit list's head chunk pointer
- `it_head_ephid`: the in-transit list's head chunk ephemeral ID + 1
- `it_tail_ephid`: the in-transit list's tail chunk ephemeral ID
- `rt_head`: the retransmit list's head chunk pointer
- `rt_tail`: the retransmit list's tail chunk pointer
- `rt_head_ephid`: the retransmit list's head chunk ephemeral ID + 1
- `rt_tail_ephid`: the retransmit list's tail chunk ephemeral ID
- `rt_count`: the retransmit list chunk count

The central arbitration module does not need to keep track of the in-transit list's tail chunk pointer. This is because upon insertion of a chunk, given the fixed size chunk allocations, the next chunk allocation can already be reserved. Thus the address of the to-be inserted chunk's next chunk pointer is known even before its successor is written

into memory, and it does not need to be manipulated upon insertion of a new chunk. Furthermore, due to the strictly and monotonically increasing nature of the in-transit list's ephemeral IDs, the count of elements within the in-transit list can be calculated as `it_tail_ephid − it_head_ephid`. This is not possible for the retransmit list, as the ephemeral IDs of elements in this list are strictly but not monotonically increasing.

The following operations, implemented through individual modules, each operate directly on the DRAM memory region. However, before commencing any operation, they must generally request permission at the central arbitration module. Along with this request, these modules provide all relevant data describing the requested operation. If this request is granted, the respective module is obliged to inform the central arbitration module of the successful or erroneous completion of the announced operation. Using this information, the central arbitration module is able to ensure that the centrally managed state remains consistent with the in-memory representation of the data structure and can prevent colliding operations from executing in parallel.

**Input and Initialization** Following Section 5.1.3.4, to initialize the linked fragmented buffer structure in memory, the `valid` flag for every possible chunk start address has to be asserted. Given that the module responsible for inserting chunks into the data structure has the required DRAM memory interfaces to write to memory, these two operations can be combined. The initialization port provided to the user of the linked fragmented buffer features two signals: an input signal `start` and an output signal `done`. It is sufficient to assert `start` for a single cycle, which will start the initialization procedure. The module will assert `done` for a single cycle when it is initialized.

The data input interface is more complex. This is because the module needs to accept payload and metadata. When a chunk is completed, the input interface needs to provide certain properties of the written chunk for them to be passed on to the transport protocol, such as the chunk's ephemeral ID and address in memory. Finally, the input interface must accept data words of width $n$ bit (e.g., 32 bit). However, the underlying DRAM interface may be significantly wider, as illustrated in Section 6.2.1. This implies that, to sustain bursts of incoming data from the data source, the input interface may need to accept up to $c$ data words in a single cycle, with $1 \leq c \leq \left\lceil \frac{n}{N} \right\rceil$ for an $N$ bit-wide DRAM interface. In turn, this requires the module to not only accept an $N$ bit data signal, but furthermore a signal indicating how many valid and left-aligned data words of $n$ bit this data signal holds.

**Peek Head** The *peek head* module performs the duties of *in-transit head peeking* as described in Section 5.1.3.2. Specifically, it observes three signals provided by the central arbitration module: the current in-transit list head pointer, the in-transit list element count, and the retransmit list element count. To provide the employed ARQ mechanism with accurate information about the oldest unacknowledged chunk in the buffer, this module invalidates any data previously provided over its interface as soon as the in-transit list header pointer changes. Subsequently, it issues a read request for the updated address. Once that read request has been completed, it pro-

vides metadata such as the chunk's transport protocol ID, ephemeral ID, and the insertion timestamp on its interface.

The peek head module must not indicate read header data to be valid when the in-transit list element count is 0, meaning that the observed head pointer is invalid, or when the retransmit list element count is not 0, meaning that there is at least one known-lost segment to be retransmit before any ARQ retransmissions of unacknowledged chunks.

**Readback** The *readback* module implements the logic required to retrieve data stored in the linked fragmented buffer, either in response to a retransmission request through the ARQ mechanism or in order to retransmit known-lost data from the retransmit list. As such, it provides a simple interface: by observing the central arbitration module's current retransmit list element count, it provides a signal indicating whether a chunk of the retransmit list is available to read. For the in-transit list, this is already indicated through the *peek head* interface.

To request reading back a chunk the provided `ready` signal should be asserted. In response to this signal, the implementation will latch onto a chunk to read back, where priority is given to chunks contained in the retransmit list. If neither the retransmit nor the in-transit list contain elements to read, the implementation indicates this condition by asserting a `stall` signal. If, in a single clock cycle, `ready` is asserted but `stall` is not, the readback request is accepted. The module will continue to output up to $c$ times $n$ bit data words per cycle, analog to the *input* module, and asserts an `end` signal on the last data word. In addition to that, after the full chunk header has been read, it provides the respective header values such as the ephemeral ID, transport protocol ID and payload data length for a single cycle, in which a `header_valid` signal is asserted.

Although this module has a conceptually simple interface, an implementation making efficient use of the DRAM interfaces as described in Section 6.2.1 is not trivial. To utilize the full bandwidth of the DRAM memory and controller, multiple pipelined read operations must be staged. Each of these staged requests will require multiple cycles until the requested data is provided (request latency). However, because a chunk contains its payload length specification in the in-memory header, the exact number of DRAM words to read can only be determined after the chunk header has been read. If the implementation were to wait for the header until issuing further read requests, this would significantly decrease the readback performance. Thus an efficient readback implementation must opportunistically insert potentially gratuitous reads before retrieving the chunk header and keep track of how many reads have been issued. After the chunk header is read, the number of DRAM words to be read can be determined. It may happen that the implementation has issued more read requests than would be required to read all payload data; in this case, the implementation has to also accept any read data in response to a gratuitous read operation.

**Acknowledge** Finally, the remaining module is the one responsible for processing acknowledgments as per Section 5.1.3.2. It is also the most tightly integrated with the central arbitration module, cooperating to transform the maintained list structures and providing a consistent view of the data structure for other modules while an acknowledgment is being processed.

To start an acknowledgment operation, this module is provided with all data present in acknowledgment messages, namely the leftmost chunk's predecessor address, the leftmost chunk address & ephemeral ID, as well as the rightmost chunk address and ephemeral ID. This information is also provided to the central arbitration module. If the acknowledgment is not entirely contained in the in-transit list or its leftmost chunk is currently being read back, the acknowledgment operation is refused. If the acknowledgment operation is accepted, the readback module will not be granted any requested read of the in-transit list head until the acknowledgment has been processed.

When the central arbitration module grants the request, the acknowledgment module continues by reading the next chunk pointer field of the rightmost chunk's header and provides this to the arbitration module. If the retransmit list is not empty and the acknowledgment's leftmost chunk does not coincide with the in-transit list's head chunk, chunk(s) left of the acknowledgment need to be appended to the retransmit list. In this case, the arbitration module will request the readback module to rewrite the retransmit list's tail chunk's next chunk pointer to point to the current in-transit head element (refer to Figure 5.4 for a visual representation of this process). Following this step, the central arbitration module modifies its internal state to represent the acknowledgment accordingly, completing the acknowledgment process.

Finally, the linked fragmented buffer implementation further exposes a *statistics* interface, which provides information about the number of chunks in the in-transit and retransmit list, respectively. If more information about the buffer contents is required, this interface could be extended to include additional information, such as the average inserted chunk length or sparseness of the ephemeral ID space in the retransmit list.

### 6.2.3. FIFO-based Memory Access Hazard Detector

While the architecture and implementation strategy described in the previous section significantly reduces the linked fragmented buffer implementation complexity, it has an inherent issue when combining it with the DRAM interfaces described in Section 6.2.1. Recall that the DRAM interface is pipelined for efficiency reasons, meaning that multiple operations can be enqueued and pending before their writes or reads are executed, respectively. This can cause severe inconsistencies in practice, for instance, when writing a chunk and reading it back shortly afterward. When a chunk is inserted, its header DRAM word is written last to correctly reflect the amount of payload data contained in the chunk. Reading back a chunk from memory, however, reads the header first in order to determine the number of subsequent DRAM words to read, using a different DRAM port. The readback

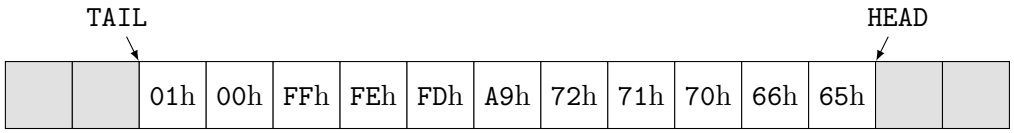
operation will retrieve outdated data if the read is executed while the header write is still pending. For this reason, a synchronization mechanism between different DRAM ports is required.

In the context of central processing unit (CPU) design, when some memory is read before a write has been committed to it in a sequence of pipelined instructions, this is referred to as a *read after write* (RAW) data hazard. The hazard conditions of concern for this thesis are not within a single DRAM port's enqueued operations, but rather between the enqueued operations of DRAM ports in conjunction with updates of the central arbitration state. When a module performs an operation updating values in DRAM and also causing an update to some central arbitration module registers, the register updates will be executed before the DRAM writes are committed to memory. These register updates can trigger different operations executed in parallel. When these parallel operations read memory pending an uncommitted write, this memory is inconsistent with the central arbitration registers. To avoid this hazard, it is sufficient to check whether a given address to be read is already pending a write operation (referred to as a *hazardous address*), and wait until all writes affecting this address have been committed. Such a hazard detection scheme, efficiently implementable within an FPGA and compatible with the access patterns of this thesis, shall be devised and implemented.

A simple but inefficient implementation of such a hazard detection mechanism is to use a FIFO queue for addresses pending a write operation, stored in a circular buffer structure. This structure's head and tail pointers can be used to determine whether a given address element is valid. Then, to test if a given address is hazardous, it needs to be compared with every valid element of the address FIFO. Such a comparison of multiple multi-bit signals consumes a considerable amount of FPGA resources, with the resource consumption increasing linearly over both the compared addresses and their bit width. Thus, a more efficient method of testing for write hazard conditions needs to be developed.

Any hazard detection algorithm must ultimately keep track of addresses subject to pending write operations. However, when making certain assumptions about the patterns of memory accesses, the number of comparisons can be reduced to achieve constant timing and logic complexity. In the following text, wrapping address semantics are assumed, meaning that an address specification $x$ for an element in memory $M$ with $n$ elements describes the memory location at the index of the corresponding least residue system element $x_M \equiv x \pmod{n}$. If accesses are guaranteed to be sequential, a span over the addresses contained in the FIFO queue can be described through addresses $a$ and $b$: upon insertion of an address $x$, if $a = b$, set $a = x$. In either case, set $b = x + 1$. When removing an element $x$, if $x = a$, set $a$ to the next element to be removed $x_{\text{next}}$. Now, for any address $x \in [a, b)$, a write may be pending, while for any address $x \notin [a, b)$ it can be guaranteed that there is no write pending. However, this scheme only works when addresses are guaranteed to be sequential, and no address $x \in [a, b)$ is inserted in the FIFO queue. Figure 6.2 provides a visual representation of this.

For write accesses, because of the memory allocation mechanism outlined in Section 5.1.3.4, the access pattern induced by the linked fragmented buffer should be sequential on a chunk granularity. However, when writing a single chunk, its payload data is written to




Hazardous Region Start: `65h` (inclusive), End: `02h` (exclusive)

Figure 6.2.: Illustration of a simple FIFO queue based hazard detection mechanism. By tracking insertion and removal of sequential (but wrapping) addresses pending a write operation in a FIFO queue, a hazardous region based on a *start* and *end* address can be tracked. This scheme does not support insertion of an address $x \in [start, end)$.

memory before its header, located at the start of the chunk in-memory address, violating the constraints imposed by the above mechanism. This issue can be circumvented, as any hazard condition is fundamentally only occurring when accessing header information: for instance, when writing a chunk, because its header is written last, all payload data are guaranteed to be committed to memory before the header. Reading a chunk starts with the header, and thus if it is no longer hazardous, the payload data is also committed. Thus it is sufficient to track write accesses to headers.

Still, the allocation scheme may cause a write to be issued within the span of potentially hazardous addresses as defined by the above mechanism. Based on the above algorithm, this thesis proposes an algorithm to handle these cases gracefully by marking the entire memory as hazardous, until the count of addresses in the FIFO queue reaches 0. In that, a fundamental assumption of the presented algorithm is that such writes into the span of potentially hazardous addresses are exceedingly rare, which holds given the employed memory allocation scheme. The algorithm requires the following control data to be maintained:

- `address_count`: a counter of addresses pending writes, contained in the observed FIFO queue (initially 0).
- `hazard_start`: the start address of the hazardous region (initially 0).
- `hazard_end`: the end address of the hazardous region (initially 0).
- `hazard_all`: a Boolean flag whether the entire memory must be treated as hazardous (initially deasserted).
- `delayed_source`: the previous address removed from the FIFO queue.

The hazard detection mechanism further observes which addresses enter and leave the FIFO queue of addresses pending writes. Based on this information and its maintained state, it implements the following algorithm. When an address leaves the FIFO queue, perform these operations:

1. Decrement `address_count`.
2. If `delayed_source` = `hazard_start`, set `hazard_start` to the removed address.
3. If `address_count` = 0, deassert `hazard_all`.

4. Set `delayed_source` to the removed address.

Furthermore, when an address enters the FIFO queue, perform the following operations:

1. Increment `address_count`.
2. If `hazard_all` is asserted, or the inserted address is contained in [`hazard_start`, `hazard_end`), assert `hazard_all` and set `hazard_end` to `hazard_start`.
3. If `hazard_all` is deasserted, set `hazard_end` to the inserted address + 1.
4. If `address_count` is equal to 1 (meaning that it was 0 before inserting the current address), set `hazard_start` to the inserted address.

Then, to test whether a given address is potentially hazardous, it is sufficient to perform the following checks:

1. If `address_count` = 0, the address is not hazardous.
2. If `hazard_all` is asserted, the address is potentially hazardous.
3. If the address is in [`hazard_start`, `hazard_end`), it is potentially hazardous, otherwise it is not hazardous.

Note that these tests are independent of the employed FIFO queue depth and multiple of these tests can be run in parallel, based on the same control data. Thus, this algorithm appears suitable for the purposes of this thesis.

## 6.3. HELIX Protocol Implementation

With the linked fragmented buffer as the core component of HELIX implemented, the remainder of this chapter shall outline the implementation of the surrounding network protocol. Figure 6.3 illustrates the overall structure of the proposed HELIX hardware architecture.

Because of time constraints, it was not possible to provide a proof of concept implementation of the ARQ mechanism responsible for automatic retransmissions, as well as the TIMELY congestion control algorithm, as part of this thesis. Thus this section will focus on the implementation of the RTT estimation mechanism. Nonetheless, various parts of HELIX already work, such as the transmission of segments, generation of acknowledgments and data reassembly at the receiver, processing of acknowledgments and manually triggered retransmissions.

### 6.3.1. Round-trip Time and Gradient Estimation

An estimation of the round-trip propagation delay between the sender and receiver is essential for the implementation of an efficient ARQ mechanism and can further be used by congestion control algorithms such as TIMELY. There are two different strategies to

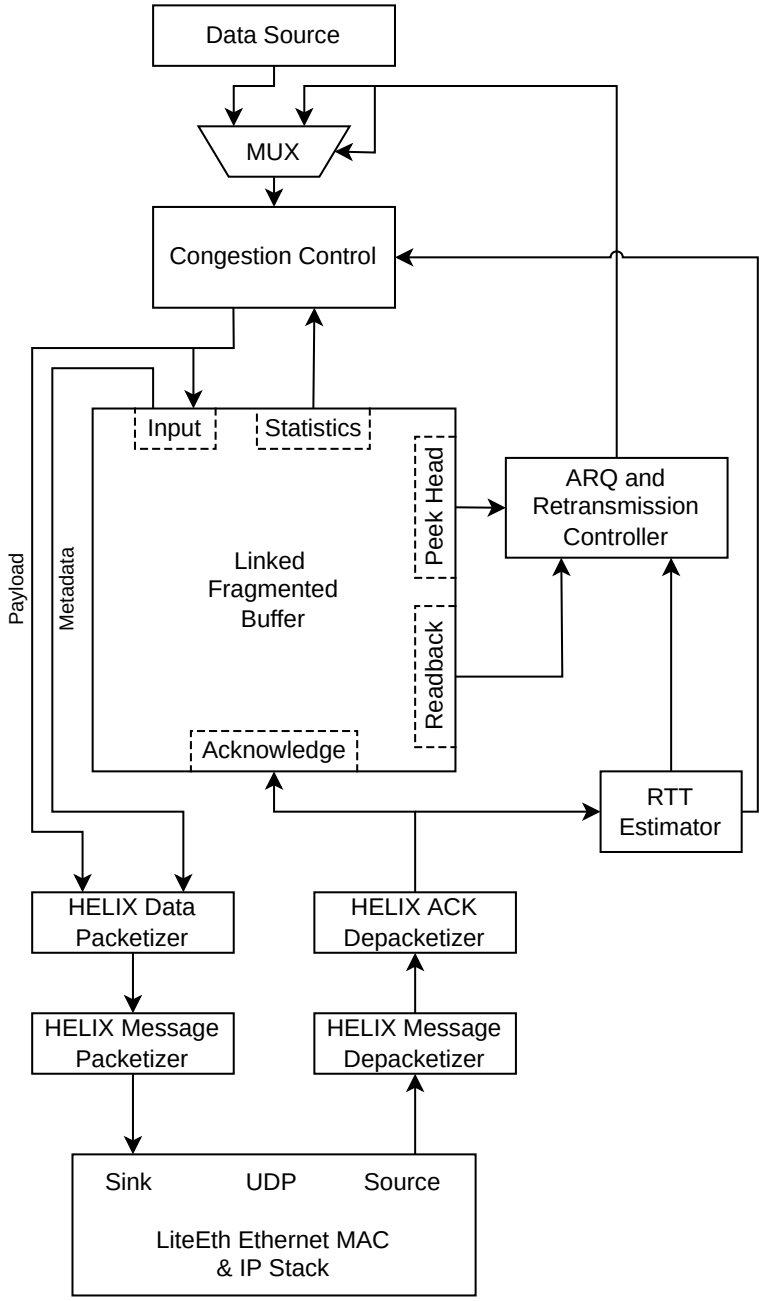



Figure 6.3.: Abstract implementation architecture of HELIX, outling hardware modules responsible for implementing specific parts of the HELIX protocol and their interactions.

retrieve an estimate of the current RTT. First, the sender can maintain a mapping of unique packet identifiers (such as the ephemeral IDs) and the transmission time of the respective packets. When it receives an acknowledgment concerning this packet identifier, an estimate of the RTT is given by the difference between the acknowledgment reception time and packet transmission time. However, because HELIX's acknowledgments can cover an entire range of segments (and thus packets), it is important to calculate the timestamps using the rightmost ephemeral ID supplied as part of the acknowledgment, as this packet will have resided at the receiver for the least amount of time. A different and likely more accurate method to measure the RTT, which does not depend on timestamps maintained at the transmitter, is to include the transmission time $t_0$ with a packet. When the receiver receives this packet, it can record the reception timestamp $t_1$. The acknowledgment message will then contain the transmission time of the rightmost chunk $t_0$ and the interval $t_r = t_2 - t_1$, where $t_2$ is the approximate time this acknowledgment will be sent at. Upon reception of the acknowledgment, the transmitter can estimate the RTT as $\texttt{rtt} \approx t_3 - t_0 - t_r$, where $t_3$ is the time of reception of the acknowledgment. However, this requires sender and receiver to agree on a common time resolution, at least for $t_r$. Furthermore, the sender and receiver may not be syntonized, meaning that their clocks increment time at a different rate. If this poses to be an issue, a protocol such as PTP (IEEE 1588) can synchronize transmitter and receiver clocks.

Regardless of which method is used to obtain such RTT measurements, a single measurement can be severely distorted due to network artifacts or scheduling behavior of the receiving host, specifically if the receiving host is not capable of taking packet reception and transmission timestamps in the NIC. A particularly simple and efficient method to correct for such outliers in the data is to use a moving average filter over $n$ RTT samples $x$ with $\overline{x} = \frac{1}{N}\sum_{i=0}^{n-1} x_{-i}$ [44]. In an FPGA design, this can be represented using a shift register of depth $n$ and an accumulator $a$. For each new sample, calculate $a_i = a_{i-1} + x_0 - x_{-n}$. The RTT is provided through $\overline{x} = \frac{a}{n}$. If $n \in \{2^m \mid m \in \mathbb{N}_0\}$, this division can be expressed as a left-shift operation by $\log_2 n$ bit.

Congestion control algorithms such as TIMELY utilize the RTT to control the transmission rate of data. Thus they are not interested in any absolute measurements of the observable network propagation delay but rather its rate of change (gradient) [35]. The same argument concerning the precision of a single measurement holds as above; thus, to determine the RTT's derivative, the averaged value $\overline{x}$ can be used as well. To obtain the rate of change of $\overline{x}$ over time $\frac{\mathrm{d}\overline{x}}{\mathrm{d}t}$, for each new sample of $\overline{x}_i$ produced, it shall be approximated by the average of the difference of the current and previous samples $\overline{x}_i - \overline{x}_{i-1}$. The same moving average filter as described in the previous paragraph can be applied to obtain an average of this gradient.

This approximation appears to work reasonably well in practice. In an ns-3 based simulation similar to that of Figure 5.6, the developed hardware implementation retrieves the measured RTT values as observed by the receiver in response to varying degrees of link utilization. The raw RTT measured within the deterministic simulation (blue plot) has been mixed with normally distributed random noise, with a mean around $\mu = 1\,\mathrm{ms}$ and a standard deviation $\sigma = 500\,\mu\mathrm{s}$. The output of the moving average filter producing $\overline{x}$ is shown in black, following the raw RTT scale (blue). The unit- and dimensionless output

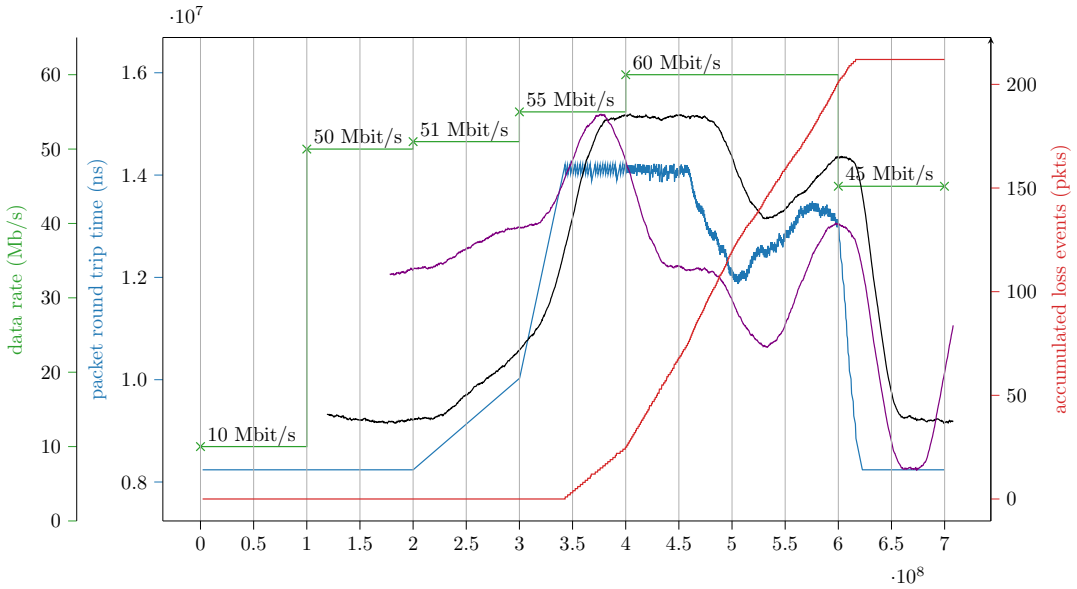



Figure 6.4.: Estimation of the RTT and its gradient at the receiver, based on explicit feedback through acknowledgments. This data has been obtained through a deterministic ns-3 simulation. The blue plot shows the raw RTT observed at the receiver through acknowledgment feedback. The black plot shows the RTT estimator's averaged RTT, based on the raw measurements mixed with normally distributed random noise, following the blue plot's scale. The dimensionless estimated gradient based on this average value is shown in purple.

of the averaged gradient of this signal is shown in purple. While both the averaged RTT measurement as well as its gradient show a delay compared to the raw RTT measurements, they compensate the introduced noise well and give a good and stable estimation of the RTT along with its rate of change.

# 7. Evaluation

Given the novelty of many of the concepts introduced in this thesis, it is important to verify their correctness and applicability to the problem at hand. Furthermore, the developed concepts should satisfy the performance metrics as required by the defined target application area of the system model.

Unfortunately, given the time constraints of this thesis, it was not possible to provide proof-of-concept implementations of some components essential to the operation of the designed protocol, specifically the ARQ mechanism and congestion control algorithm. For this reason, end-to-end measurements of the developed solution are not possible. Nonetheless, the system aspects for which no implementation exists are based on well-established concepts and algorithms. Therefore, we focus our performance evaluation on the novel concepts proposed in this thesis to show that they meet their performance targets and do not impose an upper bound on HELIX's performance.

Thus, this chapter will first present methods to test the individual component's correctness, emphasizing on testing the linked fragmented buffer. Following this, a benchmark of the linked fragmented buffer in a worst-case sustained load scenario is demonstrated.

## 7.1. Verification of Hardware Modules: Methodology

For any designed software or hardware component, it is important to test and verify that it behaves as expected. There are multiple approaches to validate that some software or hardware system adheres to a given specification. Fundamental to design verification is that in order to verify a design, one must devise design approach from the underlying component specification, different than that of the implementation. If the verification is simply carried out as a reimplementation of the same algorithms and mechanisms of the implementation, this does not necessarily verify that the implementation to be tested complies with its specifications [45]. Verification can be used to uncover both implementation errors, as well as noncompliance with a formal specification, depending on what information is available during verification and at what level the verification is carried out. Generally, one can distinguish between two types of verification methodologies: simulation-based verification and formal verification. Formal verification validates adherence to a set of formally defined rules and exhaustively checks whether these rules are upheld. In contrast, simulation-based verification stimulates a design with certain input data (test vectors) and validates that produced output data are correct through a so-called *test bench* [45].

For the intents and purposes of this thesis, the provided components shall be tested using a simulation-based verification approach. This is motivated by the high complexity involved in defining a formal specification for complex components, such as the linked fragmented buffer. Simulation-based verification can test compliance with a specification, although without formal verification no definitive statement can be made about whether a component is fully compliant with such a specification. However, simulation approaches in which test vectors are generated randomly—according to a set of rules (*fuzzing*)—can excercise many conditions implemented in such a component without tedious specification of the individual test vectors. Despite not being able to make formal guarantees about the design, *fuzzing* shall be used to verify the components developed as part of this thesis.

This section illustrates the fuzzing approach based on the devised hazard detection scheme of Section 6.2.3. Extensive fuzzing-based test benches have been developed for many of the additional components implemented throughout this thesis, including the LiteEth `Packetizer` and `Depacketizer` components. The Migen HDL provides the required infrastructure to implement test benches using arbitrary logic written in the Python programming language [42].

A test using randomly generated addresses shall be conducted to validate that the devised hazard detection algorithm can indeed detect hazardous conditions. For this, the hazard detection component is instantiated, along with a FIFO queue of addresses, as required by the component. Per test cycle, with a chance of 50%, a random address is inserted into the FIFO queue. Furthermore, with a chance of 50% and if the queue contains at least one element, the tail address is removed from the queue. Then, the implementation injects random addresses into the linked fragmented buffer test port and observes whether they are reported as hazardous. If an address is not reported as hazardous although it is contained in the FIFO queue, the module is not compliant with its specification. To end the tests eventually, the number of tested cycles is limited. If this number is reached without any error, the test is deemed successful.

However, the above tests would succeed even if the component were to report every tested address as hazardous. It must additionally be verified that in the case of sequential addresses ranging from $a$ to $b$, any address not contained in the span of $[a, b)$ is non-hazardous. Thus a second test bench will insert and remove addresses as described above. However, it ensures addresses are sequential and keeps track of the span of addresses $[a, b)$ contained in the queue. It does not insert any address $x \in [a, b)$. Then, it tests both addresses $x_\mathrm{h} \in [a, b)$, which must be reported as hazardous, and $x_\mathrm{nh} \notin [a, b)$, for which the module must not report a hazard condition.

With this basic understanding of verifying and especially fuzzing of hardware modules, we can proceeed to testing the linked fragmented buffer component.

## 7.2. Verification of the Linked Fragmented Buffer

A significant difficulty in verifying the correctness of the linked fragmented buffer component is that it provides multiple interdependent interfaces, which can all operate in parallel. Combined with the ability to store data for extended periods, a simple stream of input and corresponding output vectors is insufficient to model the module's functional specification.

Instead, the module shall be modeled as described in Section 5.1.3. However, this abstract representation should only represent the minimum required information to model the module's input and output behavior over time. The data structures employed in this model need not be efficient if this were to make the model more complex. This is in contrast to the proper implementation, which is exceedingly complex due to arbitration of operations, the requirement to interface with the LiteDRAM memory controller, bounds on the logic and time complexity of implemented operations, and parallelism of the implemented logic in general. Thus the tests implemented for this module do not verify the correctness of the end-to-end behavior of HELIX, but the adherence of the linked fragmented buffer to its specification. As such, it is further evidence (albeit not formally proven) that the proposed design of Chapter 5 does indeed work given the combination of input-, register- and DRAM-maintained data.

The central test bench maintained state is:

- `intransit_chunks`: A list of chunks contained in in-transit lists. Each element contains the respective chunk's payload data, the transport protocol chunk ID, as well as the ephemeral ID and chunk address returned upon insertion.
- `retransmit_chunks`: A list of chunks contained in the retransmit list. It holds the same type of data as contained in the `intransit_chunks`.
- `sent_chunks`: This list represents the list of *chunk message* and shall emulate a network transport. Chunks transition through this list in a FIFO fashion. Each element contains the transport protocol chunk ID, the ephemeral ID and chunk address returned upon insertion, as well as the previous chunk address and a transmission timestamp. This timestamp is used to emulate a network propagation delay.

Based on this central state, multiple test benches are defined. Each of these test benches executes independently of the others, sharing only the central state as described above. In contrast to the linked fragmented buffer logic, which by nature of FPGAs runs entirely in parallel, the test benches are evaluated sequentially per simulated hardware clock cycle, alleviating many internal synchronization and consistency concerns.

The linked fragmented buffer module is exercised through these test benches in the following ways. Note that this is not an exhaustive list of cases tested; the proper test benches test significantly more invariants.

- The *input* test bench randomly generates chunks with random payload data and a random transport protocol ID. It attempts to insert these chunks into the linked

fragmented buffer. Invariants validated in these tests include checks to ensure that an insertion operation is not refused unless the ephemeral ID span of the in-transit and retransmit lists indicate the memory might be entirely occupied. Upon successful insertion of a chunk, it is placed into the `intransit_chunks` and `sent_chunks` queue.

- The *peek head* test bench verifies that the peek head port indicates valid chunk metadata if and only if there is at least one chunk in the in-transit list and no chunk in the retransmit list. If it indicates to present valid metadata, these data are checked to correspond to the `intransit_chunks`'s head element.
- The *readback* test bench tries to read chunks from the buffer opportunistically. If the linked fragmented buffer indicates a *stall* condition, it verifies that both *intransit_chunks* and *retransmit_chunks* must be empty. If reading a chunk succeeds, it validates that this chunk must be the retransmit list's head element if `retransmit_chunks` is not empty or the in-transit list's head element otherwise. The test bench will remove the head element of the appropriate list.
- Finally, the *acknowledge* test bench observes the `sent_chunks` state. If a time $t_{\delta\text{snt}}$ has passed since the head element has been inserted, it will be removed from this list and either (randomly) dropped or counted toward the current pending acknowledgment. If no pending acknowledgment exists or this chunk can be appended to a pending acknowledgment, an acknowledgment may not be produced immediately. If a pending acknowledgment exists and the current *received* chunk is non-consecutive to it, the pending acknowledgment will be provided to the linked fragmented buffer immediately. Finally, an acknowledgment may only be pending for a time $t_{\delta\text{ack}}$ before it is provided to the linked fragmented buffer. The linked fragmented buffer must accept this acknowledgment if its chunks are entirely contained in the in-transit list and refuse it otherwise.

This *fuzzing*-based test architecture exercises many of the defined list transmutations and state transitions of the linked fragmented buffer component eventually. The tests are written such that they can verify adherence to the abstract model presented in Chapter 5. In practice, this test architecture helped uncover many implementation inconsistencies, particularly when performing certain operations in parallel. Furthermore, multiple insufficiently handled edge cases (such as moving elements to the retransmit list when it is empty) have been detected. Over the course of multiple days, the current linked fragmented buffer module implementation has been tested with over 1.5 million randomized chunk insertions and acknowledgments or reads, respectively, without detecting an error. From this, it can be concluded that the current implementation appears to be correct and compliant with respect to the specification of Chapter 5, at least for the transitions executed by the test benches as outlined above.

## 7.3. Linked Fragmented Buffer Load Benchmarks

Although implementations of the ARQ mechanism and congestion control algorithm are required to make statements about HELIX's end-to-end performance achievable on the

available hardware, this section aims to show that the linked fragmented buffer component does not pose an upper bound to these metrics.

To achieve a *worst-case* performance benchmark of the developed component, it should be subjected to an access pattern which incurs the highest cost on its internal bottleneck. In the case of the linked fragmented buffer, because all ports operate using modules running in parallel, this internal bottleneck is the shared LiteDRAM-provided memory interface. The LiteDRAM controller must multiplex all requests of all ports onto a single stream of operations operating on the physical memory module. Thus, a worst-case access pattern for this bottleneck component should be the insertion of chunks, followed by a delayed parallel readback of all inserted chunks. For the HELIX protocol, this would correspond to loss of an entire sequence of segments, while receiving new incoming data.

The memory bandwidth achieved with LiteDRAM on the NetFPGA-SUME board with a single 4 Gbyte `MT8KTF51264` DDR3-SDRAM module, for entirely sequential accesses, is approximately 7.2 GiB/s for writing and 5.6 GiB/s for reading data. These figures have been obtained through the integrated LiteDRAM speed test.

Given the benchmark scenario described above, the linked fragmented buffer component achieves a goodput data rate of 2.9 GiB/s under a sustained read and write load. This figure is reassuring and suggests that the linked fragmented buffer achieves an acceptable level of performance on the given hardware platform, especially considering an available Ethernet link bandwidth of 10 Gbit/s.

# 8. Conclusion and Future Work

Lastly, this chapter reviews the concepts established and work done throughout this thesis. It further summarizes potential future work to continue the development and evaluation of the HELIX network protocol and system architecture.

## 8.1. Conclusion

This thesis gave an extensive overview of what reliability in the context of communication protocols entails. Based on this knowledge, protocols establishing a reliable communication channel based on an unreliable packet-oriented data transfer mechanism, such as the IP protocol, have been examined. Specific emphasis has been placed on analyzing the TCP protocol, being the most common protocol for reliable data transfer over the internet and thus representative of years of research and development. This analysis outlined concepts such as automatic repeat requests (ARQs), flow control and congestion control, and their importance in protocols aiming to efficiently and reliably exchange data over the internet.

Following the required background knowledge, the system model described the class of devices targeted by this thesis, resource constraints, interface characteristics, and the associated challenges in establishing a reliable communication mechanism within such hardware systems. The problem statement formulated the issues identified in applying the concepts of reliable transmission protocols to the devices considered for this thesis.

With the express goal of developing a reliable communication protocol and associated system concepts suitable for implementation in an FPGA-based measurement device, Chapter 5 has explored the design space of data buffering techniques at the transmitter, flow control and congestion control mechanisms and implementation concepts for the receiving host. The novel linked fragmented buffer in-transit and retransmit data management architecture is of significant importance to the protocol's performance, as well as implementation efficiency and feasibility. It combines knowledge of data structures and FPGA design constraints with assumptions of the network behavior and concepts established through preexisting protocols like TCP into an efficient and coherent architecture to buffer data at the transmitter. Specifically, integrating aspects of memory management at the transmitter into the network protocol is an innovative design aspect not yet considered in other preexisting reliable communication protocols. Based on these considerations, the design section concludes by presenting the developed protocol, called HELIX.

Finally, this thesis provides a proof-of-concept implementation of HELIX and the linked fragmented buffer component to validate their implementation feasibility and correctness and further perform benchmarks to evaluate their performance. Unfortunately, due to time constraints, certain mechanisms important to the operation of the HELIX protocol have not been implemented. This prevents obtaining end-to-end measurements. Nonetheless, tests and measurements of the linked fragmented buffer component indicate that its developed implementation is correct, adheres to the developed specification, and does not impose an upper bound on HELIX's achievable performance.

## 8.2. Future Work

While this thesis established essential concepts for the development of reliable communication protocols for hardware (FPGA) systems, it is by no means complete. First and foremost, the ARQ mechanism and congestion control algorithm must be implemented to enable testing and verifying the implementation of HELIX in an end-to-end system.

Such a complete system can then be used to further analyze the developed memory architecture, network protocol semantics, and other concepts developed throughout this thesis. With the basic novel ideas behind HELIX and the linked fragmented buffer established, further research is required to estimate their effectiveness in real-world scenarios.

For instance, the HELIX protocol could utilize interleaved forward error correction (FEC) codes to mitigate the effect of packet losses in the network. Such an approach has been shown to be effective in practice [46].

Furthermore, the heap memory allocation scheme presented in Section 5.1.3.4 could be optimized further through introduction of an additional list based structure, as suggested by Weinstock and Wulf [33].

Apart from these concrete suggestions, although HELIX is technically incompatible to TCP as standardized in RFC 793, it shares many similarities, especially with respect to its network behavior and integration of ARQ and congestion control mechanisms. Thus it can benefit from much of the research done on TCP to adapt it to certain network architectures, load scenarios, or to generally improve its efficiency.

# A. Transmission Control Protocol Connection State Transition Diagram

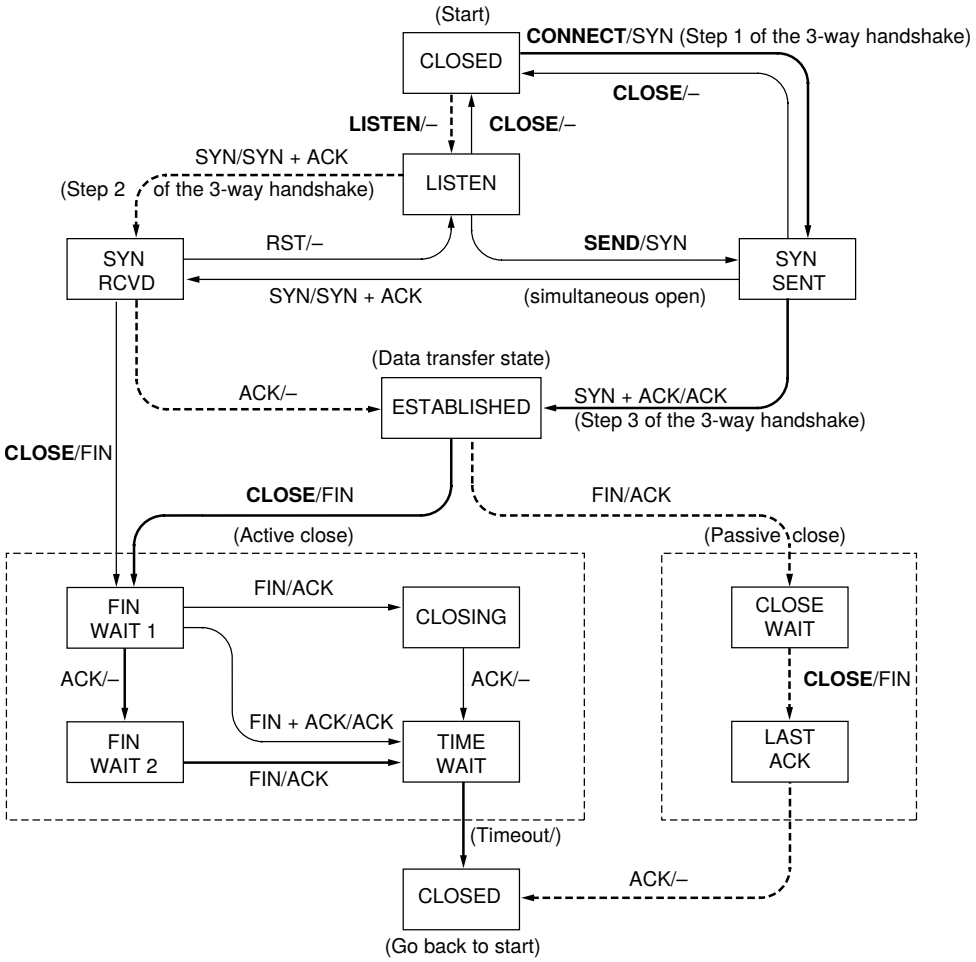



Figure A.1.: TCP connection management state machine, maintained at both ends of a TCP connection. The bold solid transitions are performed by the active-open (client) endpoint, the bold dashed transitions are executed by the passive-open (server) endpoint. Bidirectional data transfer can commence when both client and server are in the *established* state. Figure taken from *Computer Networks* by Tanenbaum and Wetherall [17, p. 564].

# Bibliography


[1] V. Jacobson and R. Braden. *TCP extensions for long-delay paths*. Oct. 1988. DOI: 10.17487/RFC1072.

[2] Patrick MeLampy. *What is Goodput – and Why It Matters*. Sept. 2021. URL: https://blogs.juniper.net/en-us/enterprise-cloud-and-transformation/what-is-goodput-and-why-it-matters (visited on 04/19/2022).

[3] *Internet Protocol*. Sept. 1981. DOI: 10.17487/RFC0791.

[4] *IEEE Standard for Ethernet*. Standard. Institute of Electrical and Electronics Engineers, 2018. DOI: 10.1109/IEEESTD.2018.8457469.

[5] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. May 2021. DOI: 10.17487/RFC9000.

[6] Jan Rüth et al. "A First Look at QUIC in the Wild". In: *Passive and Active Measurement*. Ed. by Robert Beverly, Georgios Smaragdakis, and Anja Feldmann. Springer International Publishing, 2018, pp. 255–268. ISBN: 978-3-319-76481-8.

[7] Claude Elwood Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[8] John G. Proakis and Masoud Salehi. *Digital Communications*. 5th ed. McGraw-Hill, 2008. ISBN: 978-0-07-295716-7.

[9] John Michael Spinelli. "Reliable Data Communicatin in Faulty Computer Networks". PhD thesis. Massachusetts Institute of Technology, June 1989. DOI: 1721.1/14182.

[10] J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-to-End Arguments in System Design". In: *ACM Transactions on Computer Systems* 2.4 (Nov. 1984), pp. 277–288. ISSN: 0734-2071. DOI: 10.1145/357401.357402.

[11] J.C.R. Bennett, C. Partridge, and N. Shectman. "Packet reordering is not pathological network behavior". In: *IEEE/ACM Transactions on Networking* 7.6 (1999), pp. 789–798. DOI: 10.1109/90.811445.

[12] L. Gharai, C. Perkins, and T. Lehman. "Packet reordering, high speed networks and transport protocol performance". In: *Proceedings of the 13th International Conference on Computer Communications and Networks (IEEE Cat. No.04EX969)*. 2004, pp. 73–78. DOI: 10.1109/ICCCN.2004.1401591.

[13] Alexander Zimmermann. "Das Transmission Control Protocol. Neue Wege einer modernen Loss Recovery". PhD thesis. Apr. 2012. ISBN: 978-3844009743.

[14] Wenji Wu, Phil DeMar, and Matt Crawford. "Why Can Some Advanced Ethernet NICs Cause Packet Reordering?" In: *IEEE Communications Letters* 15.2 (2011), pp. 253–255. DOI: 10.1109/LCOMM.2011.122010.102022.

[15] Michal Przybylski, Bartosz Belter, and Artur Binczewski. "Shall we worry about Packet Reordering?" In: *Computational Methods in Science and Technology* 11.2 (2005), pp. 141–146. DOI: 10.12921/cmst.2005.11.02.141-146.

[16] Kevin R. Fall and W. Richard Stevens. *TCP/IP Illustrated, Volume 1. The Protocols*. 2nd ed. Upper Saddle River, NJ, USA: Pearson Education Inc., 2011. ISBN: 978-0-321-33631-6.

[17] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. 5th ed. Boston: Prentice Hall, 2011. ISBN: 978-0-13-212695-3.

[18] Van Jacobson. "Congestion Avoidance and Control". In: *Symposium Proceedings on Communications Architectures and Protocols*. SIGCOMM '88. Stanford, CA, USA: Association for Computing Machinery, 1988, pp. 314–329. DOI: 10.1145/52324.52356.

[19] Joel Sing and Ben Soh. "Improving TCP Performance: Identifying Corruption Based Packet Loss". In: *15th IEEE International Conference on Networks*. ICON 2007. Adelaide, SA, Australia: Institute of Electrical and Electronics Engineers, 2007, pp. 400–405. DOI: 10.1109/ICON.2007.4444120.

[20] *Transmission Control Protocol*. Sept. 1981. DOI: 10.17487/RFC0793.

[21] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. *TCP Congestion Control*. Sept. 2009. DOI: 10.17487/RFC5681.

[22] R. Jain and K.K. Ramakrishnan. "Congestion avoidance in computer networks with a connectionless network layer: concepts, goals and methodology". In: *Proceedings of the 1988 Computer Networking Symposium*. 1988, pp. 134–143. DOI: 10.1109/CNS.1988.4990.

[23] Sally Floyd et al. *TCP Selective Acknowledgment Options*. Oct. 1996. DOI: 10.17487/RFC2018.

[24] Eddie Kohler, Mark J. Handley, and Sally Floyd. *Problem Statement for the Datagram Congestion Control Protocol (DCCP)*. Mar. 2006. DOI: 10.17487/RFC4336.

[25] Randall R. Stewart. *Stream Control Transmission Protocol*. Sept. 2007. DOI: 10.17487/RFC4960.

[26] Michael A. Ramalho et al. *Stream Control Transmission Protocol (SCTP) Partial Reliability Extension*. May 2004. DOI: 10.17487/RFC3758.

[27] Józef Kalisz. "Review of methods for time interval measurements with picosecond resolution". In: *Metrologia* 41.1 (Dec. 2003), pp. 17–32. DOI: 10.1088/0026-1394/41/1/004. URL: https://doi.org/10.1088/0026-1394/41/1/004.

[28] Igor Shavrin. *Pushing FPGA-based Photon Counting to the Limits. High Bandwidth and High Resolution*. Internal Presentation. Swabian Instruments GmbH, Oct. 2, 2019.

[29] Harald Devos et al. “Constructing Application-Specific Memory Hierarchies on FPGAs”. In: *Proceedings of the 2011 Conference on Transactions on High-Performance Embedded Architectures and Compilers III*. Vol. 6590. Berlin, Germany: Springer-Verlag, 2011, pp. 201–216. DOI: 10.1007/978-3-642-19448-1_11.

[30] Helali Bhuiyan et al. *TCP Implementation in Linux. A Brief Tutorial*. Tech. rep. University of Virginia, 2008.

[31] Dinesh P. Mehta. “Basic Structures”. In: *Handbook of Data Structures and Applications*. Ed. by Dinesh P. Mehta and Sartaj Sahni. Boca Raton, FL, USA: Chapman & Hall/CRC, 2005, pp. 2-1–2-16. DOI: 10.1201/9781315119335.

[32] Paul R. Wilson et al. “Dynamic storage allocation: A survey and critical review”. In: *Memory Management*. Ed. by Henry G. Baler. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116. DOI: 10.1007/3-540-60368-9_19.

[33] Charles B. Weinstock and William A. Wulf. “An Efficient Algorithm for Heap Storage Allocation”. In: *ACM SIGPLAN Notices* 23.10 (Oct. 1988), pp. 141–148. ISSN: 0362-1340. DOI: 10.1145/51607.51619.

[34] John Nagle. *Congestion Control in IP/TCP Internetworks*. Jan. 1984. DOI: 10.17487/RFC0896.

[35] Radhika Mittal et al. “TIMELY: RTT-Based Congestion Control for the Datacenter”. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM ’15. London, United Kingdom: Association for Computing Machinery, 2015, pp. 537–550. DOI: 10.1145/2785956.2787510.

[36] Robert T. Braden. *Requirements for Internet Hosts - Communication Layers*. Oct. 1989. DOI: 10.17487/RFC1122.

[37] Mark Allman. “On the Generation and Use of TCP Acknowledgments”. In: *SIGCOMM Computer Communication Review* 28.5 (Oct. 1998), pp. 4–21. ISSN: 0146-4833. DOI: 10.1145/303297.303301.

[38] Anders Rundgren, Bret Jordan, and Samuel Erdtman. *JSON Canonicalization Scheme (JCS)*. RFC 8785. June 2020. DOI: 10.17487/RFC8785.

[39] Stuart Cheshire. *TCP Performance problems caused by interaction between Nagle’s Algorithm and Delayed ACK*. May 2005. URL: http://www.stuartcheshire.org/papers/NagleDelayedAck/index.html (visited on 04/19/2022).

[40] Noa Zilberman et al. “NetFPGA SUME: Toward 100 Gbps as Research Commodity”. In: *IEEE Micro* 34.5 (2014), pp. 32–41. DOI: 10.1109/MM.2014.61.

[41] Florent Kermarrec et al. “LiteX: an open-source SoC builder and library based on Migen Python DSL”. In: *Computing Research Repository* abs/2005.02506 (2020). arXiv: 2005.02506. URL: https://arxiv.org/abs/2005.02506.

[42] M-Labs Limited, ed. *Migen 0.8.dev0 documentation*. URL: https://m-labs.hk/migen/manual/ (visited on 04/17/2022).

[43] Steve Heath. *Embedded Systems Design*. 2nd ed. Oxford, UK: Newnes, 2002. ISBN: 978-0-7506-5546-0.

[44] Steven W. Smith. *The Scientist and Engineer’s Guide to Digital Signal Processing*. San Diego, CA, USA: California Technical Publishing, 1997. URL: http://www.dspguide.com.

[45] William K. Lam. *Hardware Design Verification. Simulation and Formal Method-Based Approaches*. Upper Saddle River, NJ, USA: Pearson, 2005.

[46] Xunqi Yu et al. "A Model-Based Approach to Evaluation of the Efficacy of FEC Coding in Combating Network Packet Losses". In: *IEEE/ACM Transactions on Networking* 16.3 (2008), pp. 628–641. DOI: 10.1109/TNET.2007.900416.

# Declaration of Authorship

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted hard copies.

.....
Place, Date, Signature