Otto-von-Guericke-Universität Magdeburg
Department of Computer Science

Bachelor Thesis

Improving Packet Processing Speed on SCION Endhosts

Author:

Johann Peter Wagner

14.09.2021

Advisors:

Prof. Dr. David Hausheer
Institute For Intelligent Cooperating Systems

M. Sc. Marten Gartner
Institute For Intelligent Cooperating Systems

Wagner, Johann Peter:
Improving Packet Processing Speed on SCION Endhosts
Bachelor Thesis, Otto-von-Guericke-Universität Magdeburg, 2021.

Abstract

This work solves many performance problems in the packet processing speed of SCION. For this purpose, we use XDP as a growing technology and evaluate to what extend modern network technologies can accelerate the end-user performance of SCION applications without changing existing APIs.

The current SCION reference implementation has two problems which we cover in this work: The Dispatcher as a central element, that does not perform well under high load, and the packet processing in the client library, which leads to packet loss due to low processing throughput under high load.

We solve these problems with two optimizations: The Dispatcher Bypass through the use of XDP and packet processing improvements to the client libraries used by applications. To solve the first bottleneck, we optimized the Dispatcher's data path, in which we used XDP to take over the Dispatcher's main task of forwarding packets. For the second bottleneck, we wrote an optimized version of the SCION connection with efficient packet processing that applications use to access the SCION network. This provides further performance benefit in combination with the previous optimization.

We evaluate our optimizations in a scientific scenario and a real-world scenario. In the scientific scenario, the bandwidth approximately quadrupled with the use of Dispatcher Bypass using XDP, while in the real-world scenario, the throughput tripled. Additionally, in conjunction with the optimized SCION connection, throughput could be increased sevenfold in the scientific approach, while a threefold increase was observed in the real-world scenario.

Therefore, we show that XDP is a well suited technology to increase the throughput of SCION applications without changing existing APIs. However, the full potential of this optimization can only be achieved if the client libraries allow efficient data processing.

Table of Contents

1. Introduction

1.1 Motivation

In today's world, fast Internet speeds of more than 1 GBit/s are no longer uncommon. While TCP and UDP use highly optimized network stacks to fully utilize the available bandwidth, SCION lags behind in terms of performance.

Scalability, Control, and Isolation on Next-Generation Networks (SCION) is a new generation network architecture to overcome the technical problems of the Internet as we know it today. While it solves many known technical issues, e.g. trust issues with BGP announcements and missing path control, it creates other performance challenges due to its design decisions. Since the implementation of TCP and UDP within the Linux kernel can transmit many gigabits per second and network speeds will raise [3], SCIONs reference implementation needs to overcome some performance bottlenecks limiting packet processing power. The Dispatcher, a required part of any SCION client installation, seems to be one of the biggest performance bottlenecks since it needs to forward every incoming and outgoing packet [22].

eBPF and XDP have become more stable in recent years and have been massively upgraded in recent versions of the Linux kernel. XDP provides capabilities for direct inspection, modification or dropping of the package in kernel space. This allows higher throughput rates and smaller latencies [23].

XDP can be used to take over the performance-critical part of the Dispatcher's data processing. There are also some other accessible performance optimization possible in the client libraries, but these can only be evaluated after the major performance botleneck has been eliminated.

1.2 Goal

This work covers the design, implementation and evaluation of an XDP-based approach to remove bottlenecks from the performance-critical path of SCION as well as some general performance optimizations for packet processing.

The objective of this thesis is to answer the following research question: *To what extent can modern network-related technologies, especially XDP, accelerate the performance of end-user applications in SCION without changing APIs?*

1.3 Structure

After this short introduction and motivation of the thesis topic, we will provide some background information to cover relevant knowledge for this thesis in Chapter 2. Further, we present related work in Chapter 3 to show an overview of the current state of research. Afterwards, we describe our observed performance bottlenecks in Chapter 4 and propose possible solutions in Chapter 5. We also provide an evaluation of our proposed solutions in Chapter 6. Finally, we conclude the results in Chapter 7 and outline possible next steps to optimize the SCION network stack even further in Chapter 8.

2. Background

This chapter provides required knowlegde to understand the observed performance problems and proposed solutions. It mainly contains information about SCION, eBPF, XDP and transport protocols.

2.1 SCION

Scalability, Control, and Isolation on Next-Generation Networks (SCION) is a greenfield approach to solve todays challenges with the global Internet.

The global Internet is based on two technologies, BGP and IP, that have not drastically changed over the last 25 years. Over time more and more attacks were found to disrupt these fundamentals. DDoS attacks and BGP hijacking are two of the most known attacks. One of SCION's main design goals is the elimination of many known problems with a entirely new greenfield approach [22].

SCION refers to the SCION protocol [30], as well as to the reference implementation [26].

2.1.1 Core Concepts

SCION networks consist of several Isolation Domains (ISDs). ISDs contain one or more Autonomous Systems (ASes) and are used to provide logical clustering for ASes isolating different parts of the SCION network from each other. Each ISD agrees on a trust root configuration - its own policies, keys and authorities - and can work isolated without other ISDs ensuring that broken or malicious ISDs cannot take down the whole SCION network. Each ISD is administered by a consortium of ASes, the ISD FControlcore. It exposes a public key infrastructure for the authentication of ASes, provides inter-ISD and intra-ISD path segments for routing purposes and high availablity services, e.g. RAINS and time servers. ASes must accept the given trust root configuration of the ISD and use an issued certificate to communicate with other parts of the ISDs [22].

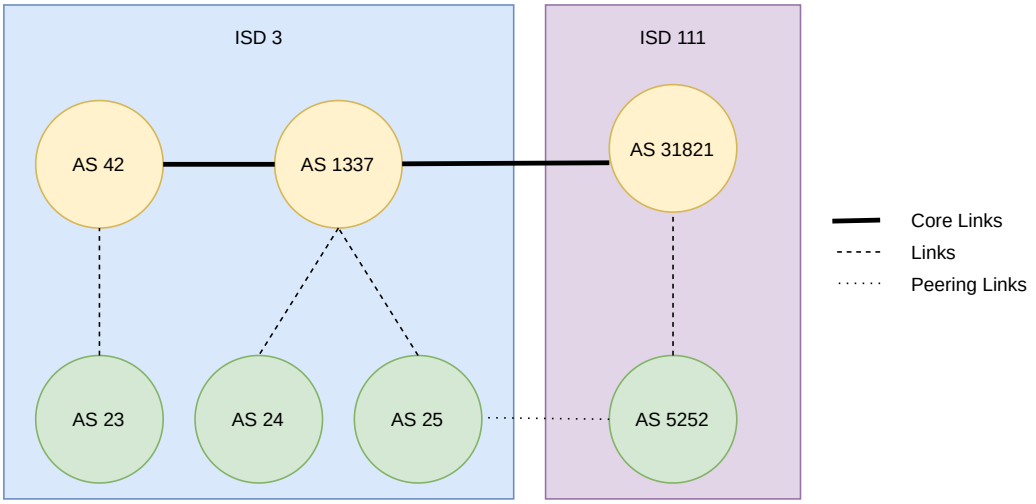

Figure 2.1: Example SCION Network

Two SCION ASes are connected to each other with one or more links. A link can be established over a physical or virtual link. ASes are generally connected to core AS as seen in Figure 2.1 between AS 42 and AS 23. Links between core ASes are called core links as seen between AS 1337 and AS 31821, links between ASes of different ISDs are called peering links as seen between AS 25 and AS 5252.

SCION distinguishes between control plane and data plane. The control plane is used to find out how to route the packet through the SCION network. The data plane is used to forward the actual packet through the SCION network. Each component of a SCION network has exactly one associated plane. Figure 2.2 shows the components of a standard SCION installation of a SCION endhost. Whereas, SCION Daemon and SCION Control Service form the control plane, and SCION Dispatcher and SCION Border Router form the data plane [22].

Figure 2.2 also shows, that each SCION host needs to run at least the SCION Daemon and the SCION Dispatcher. The SCION Dispatcher provides the data plane interface as the SCION Daemon provides the control plane interface to the SCION network for user applications. Missing one or both of the mentioned interfaces results in missing of connectivity. A SCION AS also needs to run a Control Service and a Border Router on one of its members to communicate with other SCION ASes. One SCION host can run a full SCION AS (One-Host-AS) [15].

Each SCION application needs to register at the Dispatcher to be able to send

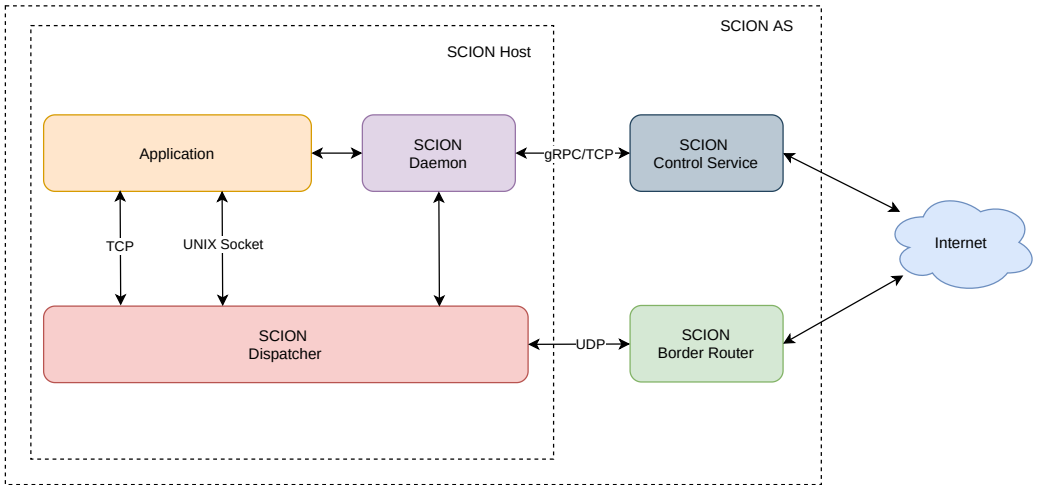

Figure 2.2: SCION Application Structure

and receive SCION packets. It also needs to use the SCION Daemon to receive routing and path information from the control plane. The SCION Control Service provides all control plane services that are needed within an AS. It handles all control plane tasks, mainly path exploration and registration and path lookups. The SCION Border Router handles AS-outgoing packets and sends them to the next AS. It also handles AS-incoming packets and sends them to the appropiate SCION Dispatcher [22].

2.1.2 SCION Packet Structure

SCION packets have a defined structure, visualized in Figure 2.3. They are wrapped in a transport protocol, e.g. UDP, which allows transport between different ASes over traditional networks. SCION packets also contain a Layer 4 protocol, e.g. UDP or TCP. They will be named SCION/UDP and SCION/TCP to differ from the transport protocol [22].

SCION headers are dynamically sized and contain the following three main areas: SCION common header, SCION address header, and SCION path header. A SCION common header contains the important information and lengths and offsets to other headers. It also contains the `NextHdr` field that indicates the Layer 4 protocol. A SCION address header contains source and destination ISDs and ASes and host adresses used by the transport protocol. The SCION path header contains the fixed path from the source AS to the destination AS. The intermediate destinations are called hops and are used by the SCION data path for forwarding the packet [22].

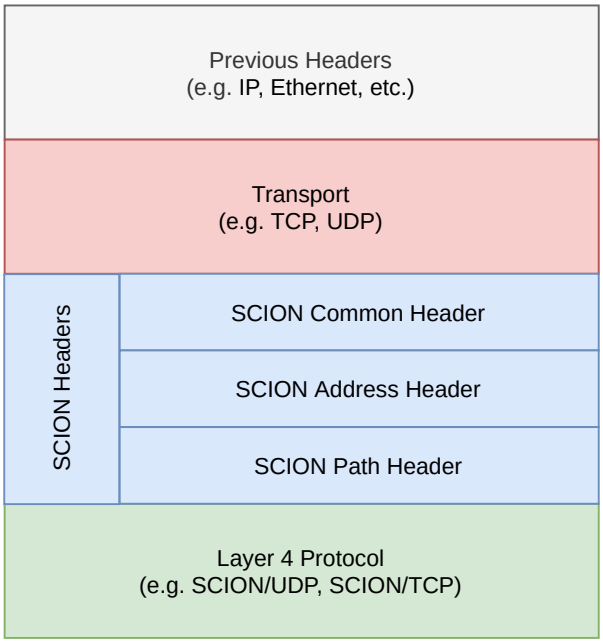

Figure 2.3: SCION Packet Structure

2.1.3 SCMP

SCION Control Message Protocol (SCMP) fulfills the same purpose in a SCION network as Internet Control Message Protocol (ICMP) within the current Internet. It mainly provides network diagnostics and error messages. Network diagnostics are needed to built network diagnostic tools for the SCION network, e.g. traceroute and ping. Error messages are used to signal problems with sent packets or network problems [22]. There is a defined set of SCMP messages that a SCION component needs to send within certain situations, e.g. no route to host, adress unreachable, port unreachable [24].

2.1.4 Dispatcher

The Dispatcher is one of the main components of a SCION endhost. Since SCION does not have any kernel integration like other network protocols, the Dispatcher is a userspace interface to achieve SCION packet encapsulation/decapsulation and other tasks typically done by the kernel integration. Thus, instead of opening a UDP socket, for example, the application connects to an existing UNIX socket provided by the Dispatcher. The Dispatcher provides the data plane interface to the SCION network using an UDP socket

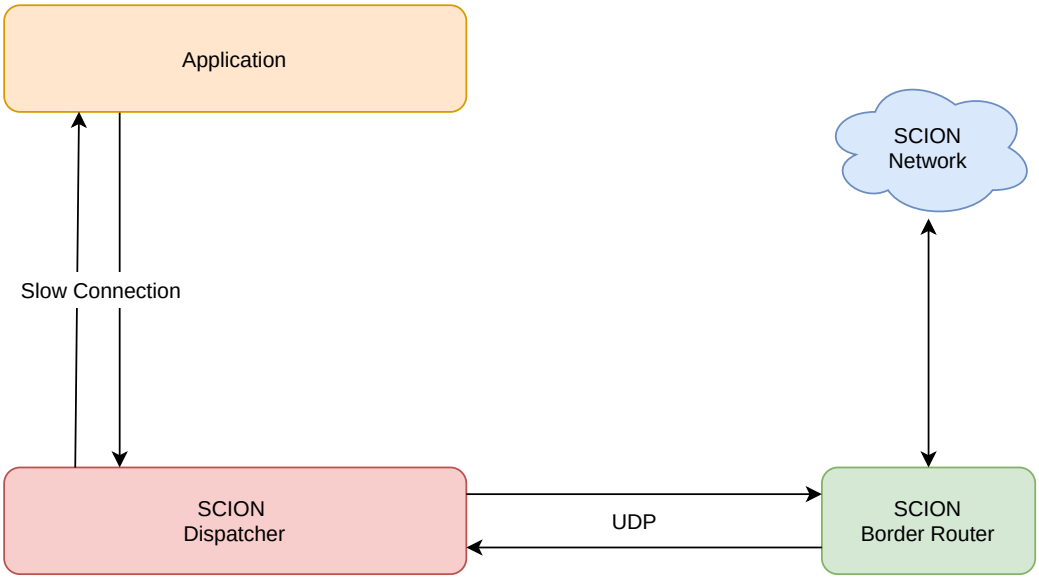

Figure 2.4: Application Connection to the SCION Network

for connectivity with the Border Router and redirects incoming packets from the UDP socket to the correct applications and sends out packets from the UNIX socket to the SCION network. The Dispatcher also plays a main role for performance since the Border Router forwards packets to the correct host but it is not aware of different applications running on the same host, mainly for performance reasons [22][26].

Figure 2.4 shows the packet flow of SCION applications. An application utilizes the UNIX socket of the Dispatcher (Slow Connection) to connect to the Dispatcher and transmit and receive data from the SCION Network. The Dispatcher acts as an relay between the SCION network and the UNIX socket and receives and sends SCION packets at its UDP socket. Having only one entrypoint for all host traffic allows the optimization that the routers do not need to be aware of the final destination port of the packet. This allows multiple performance optimizations for intermediate SCION network components between endhosts, not only for routers [22].

Each application needs to register at the Dispatcher. An application registration is a two message handshake. The application sends a listening address, a service type and an tupel of ISD identifier and AS identifier to the Dispatcher. In case of a successful registration, the Dispatcher returns an application port. The application port is saved in a routing table of the Dispatcher. It is used to forward packets to the correct applications but it does not open a port at the host machine. All application traffic is routed through the UDP socket of

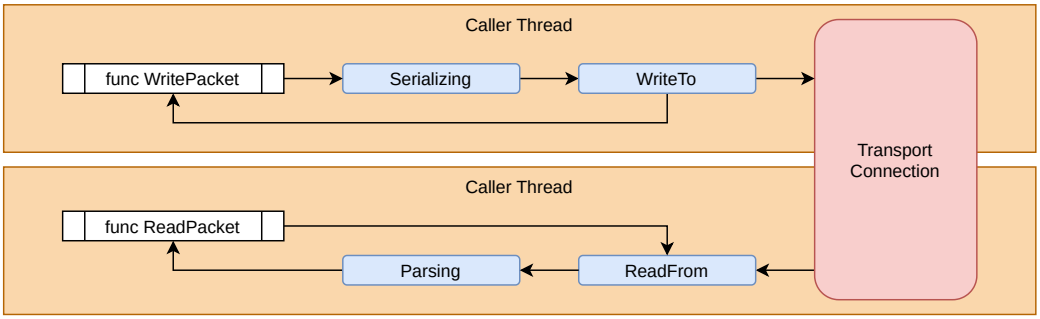

Figure 2.5: Single-Threaded Packet Processing

the Dispatcher [22].

It runs two ring buffers interally, one to move packets from the UNIX socket to the UDP socket and another for packets from the UDP socket to the UNIX socket. If there is incoming data at the UNIX socket, it gets buffered into a packet. Afterwards, it gets added to the outgoing ring buffer. The worker for the outgoing ring buffer will wrap it into a transport packet, e.g. UDP, and send it to the next hop and remove it from the ring buffer [26].

If there is an incoming packet on the UNIX socket, it gets unpacked from the transport packet. In case the SCION packet contains a SCION/UDP packet, it is added to the ring buffer and later send to the application determined from the routing table. If the unpacked packet is an SCMP packet, it is handled seperatly. If it is an unknown packet or if it is an application packet for an unregistered application, it replies with an appropiate SCMP packet [26].

Metrics are collected from packet processing that are also exposed, e.g. incoming and outgoing packets and transmitted and received bytes [26].

2.1.5 Client Packet Processing

SCION also provides a reference implementation for generating and handling SCION packets including serialization and deserialization, called `snet` [27]. It is written in Go and used in most SCION-specific scientific evaluations and for building core SCION network components. It uses a high-level packet builder that serializes packet headers to byte arrays and deserializes the other way round. It is used to convert SCION packets between raw byte arrays and an application-consumable format, e.g. Go structs [26].

Each SCION application needs to connect to the Dispatcher utilizing a

transport connection, e.g. a UNIX socket. It is used to transfer data from and to the application. In Figure 2.5 we see the packet processing in the reference implementation. If `WritePacket` gets called with a reference to a application-consumable SCION packet, it needs to be serialized to a writeable byte array and written to the underlying transport connection. Afterwards, the function returns. If `ReadPacket` gets called, it blocks until incoming data from the underlying transport connection is received. Afterwards, the read data is parsed into a provided application-consumable packet, the function returns, and the packet is ready to be consumed by the application. This all happens single-threaded within the same thread as the function call [26].

2.2 eBPF and XDP

2.2.1 eBPF

Extended Berkeley Packet Filter (eBPF) is the newer and extended version of the Berkeley Packet Filter from 1992. It gained support at Linux kernel version 3.15 while a decent feature set is available since kernel version 4.9 [6][29]. It is an instruction set for packet processing allowing interaction and modification used by Smart NICs or the Linux kernel. It can be used for packet metrics, packet routing tasks and dropping packets. It executes within kernel space omitting unneccessary copies of the network buffer [29].

Source code, written in the C language or the P4 language, can be compiled into eBPF object code, but supports only a language subset. The main missing language features are non-static global variables and dynamic loops due to impossible verification. It gets verified by a kernel verifier checking call depth, performance, license and memory access. Afterwards, the eBPF object code gets just-in-time translated to machine-specific code for the kernel interpreter or a smart NIC, if available [29].

Each eBPF program has a specific type defining the input and output values and available context. There are over 25 different supported program types for specific applications within the Linux kernel, e.g. packet dropping, metrics [29].

Maps allow persistent data storage within eBPF programs and communication with the program that attached the eBPF program. They are generic, untyped key value stores. This is realized with the usage of shared memory between kernelspace filter and userspace application. There are different types of maps

for a lot of different applications, e.g. maps and arrays. `libbpf` is used to interact with maps from the host system [29].

2.2.2 XDP

Express Data Path (XDP) is an high performance data path within the Linux kernel. It is used to view, modify or drop packets before they pass the networking stack of the kernel. XDP is a hook of eBPF and can be used by attaching a eBPF program with the type `BPF_PROG_TYPE_XDP` to one or multiple interfaces. This program is called for every incoming packet with a reference to a packet as parameter and an XDP action as return value. In between, the program can read and modify the packet and access BPF maps. The XDP action, listed within Table 2.1, defines the path of the packet through the kernel. If the packet is dropped, it is not processed by the kernel. If it is passed, the eventually modified packet gets processed by the kernel afterwards. If it is redirected in any way, another network interface will process the packets [29].

Table 2.1: XDP Actions [29] (Modified)

Value	Action	Description
0	XDP_ABORTED	Drop packet, error in processing
1	XDP_DROP	Drop packet
2	XDP_PASS	Allow further processing by the kernel stack
3	XDP_TX	Bounce packet back to interface it came from
4	XDP_REDIRECT	Redirect packet to another interface

Figure 2.6 shows an code example for an eBPF program that can be used within XDP. It receives the start pointer and end pointer from the given context and checks, if the packet length (the difference between start pointer and end pointer) is longer than 42. If yes, it passes the packet to the kernel. Else the packet is dropped.

2.3 Reliable Transport Protocols

Reliable Transport Protocols are used as an overlay protocol over SCION and other non-reliable transport protocols, e.g. UDP, to ensure data integrity.

```c
SEC("xdp") int xdp_function(struct xdp_md *ctx) {

    // Pointer to start of packet
    void *data = (void *)(long)ctx->data;
    // Pointer to end of packet
    void *data_end = (void *)(long)ctx->data_end;

    int neededOffset = 42;

    // If packet is shorter than needed, drop it.
    if(data + neededOffset < data_end) {
        return XDP_DROP;
    }

    return XDP_PASS;
}
```

Figure 2.6: XDP Code Example

2.3.1 QUIC

Quick UDP Internet Connections (QUIC) was developed by Google in 2012 and standardized in 2016. It solves performance problems of a connection-based approach, i.e. TCP, in downloading data by creating a virtual connection utilizing a connection-less transport protocol, e.g. UDP. It is used in modern browsers to transmit many small files from a server to a client without the TCP connection estabilishment overhead. It ensures packet ordering and retransmits lost packets [13].

2.3.2 PARTS

Path-Aware Reliable Transport over SCION (PARTS) is a fairly new approach and it is currenty developed by Gartner et al. at Otto-von-Guericke University in Magdeburg. Unlike QUIC, PARTS is designed to transfer large amounts of data. It adapts the throughput based on bandwidth and packet loss. Therefore, it reduces the sending rate if there is a lot of packet loss on the receiver and raises the sending rate if there is a lot of available bandwidth. It also ensures the correct transmission of the provided data chunk, since in the case a packet is lost, it will be retransmitted [19].

3. Related Work

One of the main challenges of computer science is to optimize any kind of possible problems. This applies to algorithms just as much as to transmission speeds. There is some related work on these topics, especially on performance measurements of UDP-based connections and SCION connections.

Gartner wrote his master's thesis entitled "Improving SCION Bittorrent with efficient Multipath Usage" [11], in which he evaluates the use of SCION as a transport protocol for BitTorrent and the impact of multipath capabilities on transfer speeds. He concludes that there must be a bottleneck in the SCION Dispatcher after removing the Border Router from the reference implementation in his evaluation. He was able to achieve, using Anapaya System AG's High Speed Border Router (HSR), a throughput of 500 MBit/s with one connection and a throughput of 600 MBit/s with multiple connections. After enabling jumbo frames, higher throughputs could be achieved, but much less than expected.

Neukom achieved a data transfer rate of 100 GBit/s in "High-Performance File Transfer in SCION", based on a work by Frei and Wirz [21][10]. To reach this throughput, he used the Hercules protocol and wrote his own client implementation, based on `AF_XDP`, which only uses the SCION protocol and does not use any performance critical parts of the reference implementation. He also made optimizations to the client library to achieve better throughput rates, such as using multiple sending and receiving queues and enabling multi-threading.

Apart from SCION, there is little current research on the performance of UDP connections over high-speed links, i.e. 10 GBit/s and more.

Syzov et al. developed a UDP-based transport protocol utilizing DPDK and evaluated it in their work "Custom udp-based transport protocol implementation over DPDK" [28]. Although the actual work is not directly related to SCION, they still evaluate as their baseline the UDP performance of their testbed using a very simple C program. This gives us a good reference of what to expect in terms of performance for a fast implementation. They state a sending speed of 10 GBit/s and a receiving speed of almost 7 GBit/s on a single connection. Christensen and Richter also evaluated the performance

of UDP connections in their work "Achieving reliable UDP transmission at 10 Gb/s using BSD socket for data acquisition systems", but on BSD-based systems [7]. However, they measured similar results as Syzov et al., they measured throughput of about 8 GBit/s under similar conditions.

Furthermore, although XDP is still a very new technology, there are already several papers and reports that have analyzed XDP and found comparatively high achievable bandwidth with XDP.

Miano et al. discussed in their work "Creating complex network services with ebpf: Experience and lessons learned" the application possibilities of XDP in real-world scenarios and also pointed out limitations that XDP has due to the fact that code is executed in kernel space. They also evaluated if and to what extent their solution of real-world problems with XDP have an impact on performance. They concluded that some of the limitations can be overcome and others will need to be solved in newer kernel versions. With regard to performance, very good results were achieved [17].

Hohlfeld et al. evaluated the performance of XDP in their work "Demystifying the Performance of XDP BPF". In doing so, they also addressed the various execution options, e.g., running the code on a Smart NIC. They came to the conclusion that XDP is capable of processing over 15 million packets per second, but the workload per packet should remain small, otherwise bottlenecks will occur on smart NICs [12].

Bertin did a similar analysis in their paper "XDP in practice: integrating XDP into our DDoS mitigation pipeline". They use XDP to accelerate DDoS mitigation at Cloudflare. There XDP is now used to check millions of packets per second and to drop malicious packets. Previously this was done with other kernel modules, for example `iptables`, where performance problems occurred. Cloudflare is often an early adopter of many new technologies, including XDP [2].

4. Performance Bottlenecks

The following topic describes multiple performance bottlenecks regarding the packet processing speed on SCION endhosts.

4.1 Dispatcher Performance

The Dispatcher is an often discussed component in the SCION community. It is a required and important piece of the SCION end-host stack and absolutely neccessary for connectivity of endhosts. Since SCION is not implemented within the standard networking stack of common operating systems the Dispatcher is needed to provide a network interface to SCION applications [22].

There are many problems with this approach, as the Dispatcher represents a vital part of the SCION ecosystem as shown in Chapter 2.1.4. For example, Costea states issues regarding statefulness, architecture and packet processing speed [8]. Some other SCION-related works like Gartner also experienced a huge performance bottleneck regarding packet processing speed within their evaluation [11]. They could not achieve the expected results due to a unexplainable high CPU usage of the Dispatcher. They also noticed a dependency between throughput and CPU speed [14] and a dependency between throughput and MTU [11][14]. Since raising the MTU or raising the CPU power leads to an increase of throughput, we assume a compute performance issue once per packet. We analyzed the Dispatcher in depth and can confirm their assumption.

Costea already tracked down two known performance problems [8]. The Dispatcher is a userspace program moving every packet from kernelspace to userspace, processing it, and moving it back from userspace to kernelspace. This results in a lot of expensive context switches. Also, it currently uses a non-packetized UNIX socket that requires a packetizing process.

In conclusion, the Dispatcher is conceptionally slow and need to be replaced by a faster packet forwarding solution ideally within kernelspace.

4.2 Slow Packet Handling

In the SCION reference implementation, there exists a connection implementation. It is used for example applications, other SCION applications written in Go, and for most scientific evaluations. We ran into performance bottlenecks within this implementation while designing and evaluating our solution for the Dispatcher bottlenecks.

The SCION connection implementation performs two main tasks, receiving and sending of packets, divided into two subtasks each. When receiving packets, it reads from the underlying transport connection and parses the raw packets into a processable format, for example parsing different headers and handling SCMP packets and so on. When sending packets it serializes the given packets and writes them to the underlying transport connection [26].

On the one hand, the deserialization of the packets needs a lot of time. It extracts the L4 payload from the SCION packet, which is actually only the separation of a certain part of the packet buffer. To achieve this the entire packet is parsed and unneeded parts, i.e. everything except the payload, are discarded. It also results in a lot of short-term memory usage that needs to be freed by the garbage collector. On the other hand, the serialization of the packets consumes a considerable amount of computing power. During serialization, parts that are consistent over the lifetime of the connection are regenerated unneccessarily.

Summarized, there is a (de-)serialization step and a sending or receiving step handled within the same thread that reduces the time spent handling the connection, e.g. reading or writing packets, leading to slow processing speed and dropped packets under high load.

5. Design

This chapter describes the design ideas to solve the forementioned performance bottlenecks while not touching existing core SCION core components for compatibility reasons.

5.1 Dispatcher Bypass

To solve the first of the performance problems mentioned in Chapter 4, we present here our solution approach *Dispatcher Bypass*. This is intended to solve the problem of *Dispatcher Performance*.

The SCION book proposes three different connection interfaces for applications, e.g. UDP, TCP and SCION Stream Protocol (SSP). Currently, the Dispatcher only implements a UDP connection tunneled through a UNIX socket [22]. Since the Border Routers use UDP to communicate with each other and the Dispatcher also uses an UDP connection to send and receive packets from the Border Routers, we will only focus on the UDP connection interface within this implementation.

Figure 5.1 shows two main differences to the reference implementation, the addition of a second connection (Fast Connection) and the introduction of `xiondp`, compared to the existing implementation shown in Figure 2.4. We want to introduce a second connection (Fast Connection) next to the known connection (Slow Connection). This allows us to bypass the Dispatcher. The Fast Connection is UDP based, like the connection between Dispatcher and Border Router. We create a new control binary called `xiondp` deploying an XDP program for redirecting incoming UDP packets directly at the Fast Connection without utilizing the Dispatcher.

Table 5.1: Component Combinations

xiondp	Slow Connection	Both Connections	Fast Connection
Enabled	y	y	y
Disabled	y	y	n

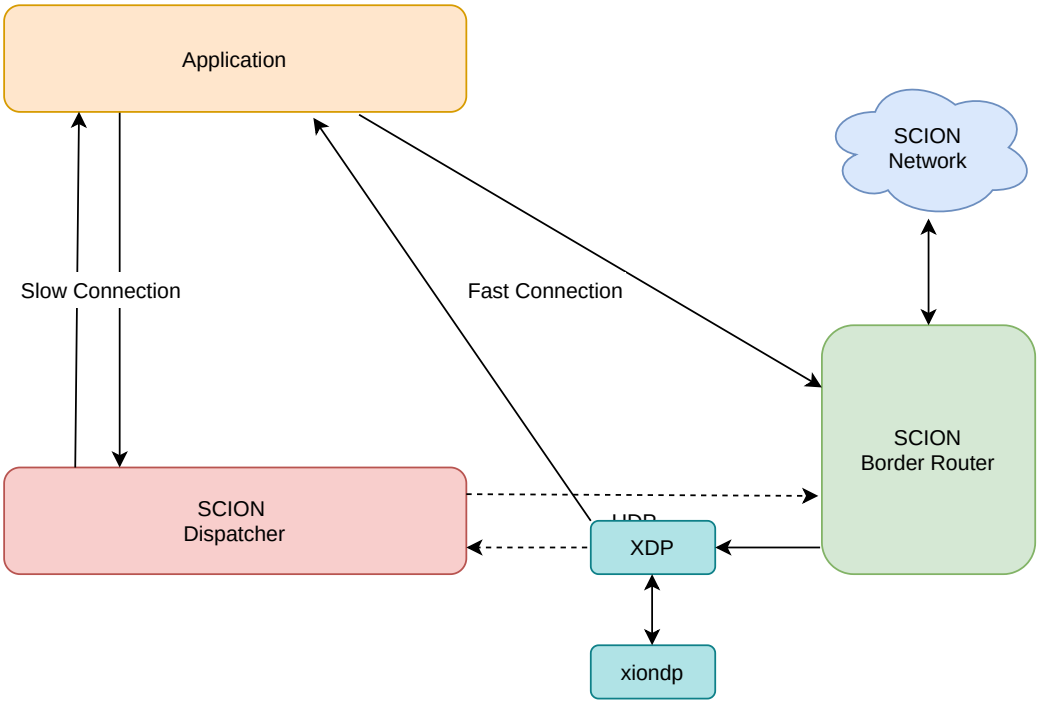

Figure 5.1: Proposed Application Connection to the Internet

We also want to create a solution that can be enabled and disabled individually per application. Table 5.1 shows the configurations possible, allowing use the existing connection, both connections or use the new Fast Connection exclusivly. Even if `xiondp` is not enabled the improved sending performance can be utilized by using both connections but packets will be received through the Slow Connection.

5.1.1 Client Library

We introduce a fundamental change to the client library. So far the client library opens a UNIX socket (Slow Connection) to communicate with the Dispatcher. After a registration handshake the same socket is used to transmit data. We introduce the fast connection bypassing the dispatcher to receive and transmit data instead of using the Slow Connection.

If the user wants to send data, we need to prepare one or more SCION packets. Before preparation, the client library needs to receive available paths to the destination from the SCION Daemon. When creating a packet, it is prepared with the selected path from the available paths. It also adds a

chunk of user data. In the reference implementation, the packet was sent via the UNIX socket to the Dispatcher. It then repacks the packet into a transport packet and sends it to the Border Router [26]. We can remove this performance-critical step by packing the SCION packet directly into a transport packet and sending it to the Border Router without utilizing the Dispatcher.

When receiving a packet, it can be received via the UDP connection or the UNIX socket. This depends on whether `xiondp` is enabled on the endhost, otherwise as before only the UNIX socket is used. To provide the best possible integration with existing systems, we should merge both incoming packet streams to provide a resilient interface to the user which may have performance issues. For this work, we decided against merging of streams for this reason, but this requires that whenever the Fast Connection is used, `xiondp` has to be enabled.

5.1.2 xiondp

`xiondp` is a word combination of SCION and XDP. It is a control binary for deploying and controlling an eBPF program utilizing XDP. The eBPF program is responsible for redirecting incoming SCION packets to the Fast Connections of the applications before processing by the Dispatcher to speed up the packet receiving speed.

The control binary attaches the eBPF program to all possible network interfaces. It also maintains a whitelist of open UDP sockets and saves this into a BPF map that can be accessed from the eBPF program.

The eBPF program is called for every incoming packet. It processes the packet and checks if it is qualified for a "fast-forward" redirection. It needs to contain an UDP header encapsulating a SCION header wrapping a SCION/UDP header. The SCION header must be adressed to the Dispatcher and the SCION/UDP header also needs to contain a whitelisted port.

Figure 5.2 shows a schematic of a potentially forwardable packet containing the mentioned headers and the process of port rewriting. If the packet can be fast-forwarded, it will be rewritten from being addressed to the Dispatcher to being addressed to the application UDP socket. Afterwards, it is passed to the remaining kernel network stack and will directly arrive at the Fast Connection within the application without an extra step via the Dispatcher. If the packet cannot be fast-forwarded, e.g. non SCION traffic or applications that do not use a Fast Connection, it will not be altered by the eBPF program. After an

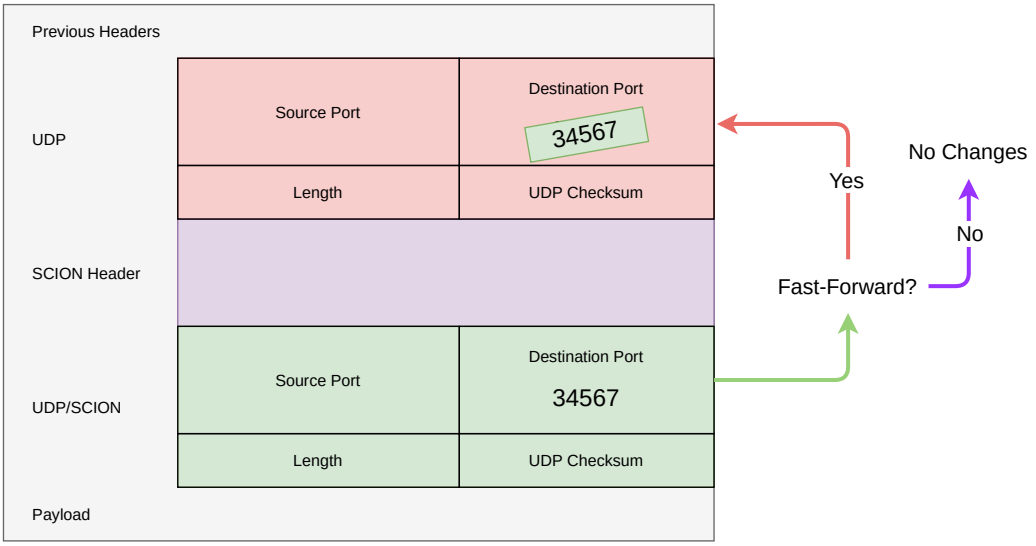

Figure 5.2: Packet Rewriting Process

unmodified SCION packet has passed through the rest of the kernel network stack, it will arrive at the Dispatcher as usual, where it will be processed and passed to the registered applications.

The eBPF program also records and classifies the processed packets in three categories, all packets, non-fast-forwarded SCION packets and fast-forwarded SCION packets. According metrics are exposed and can be used to verify the flow of packets.

5.2 Multi-Threaded Packet Handling

To solve the problem of *Slow Packet Handling*, we present here our solution approach *Multi-Threaded Packet Handling*.

Currently, the packet handling is a single-threaded implementation using the simplest possible approach to handle packets. It reads and parses the packets in the same thread and serializes and send the packets in the same thread. We want to parallelize the implementation to improve the packet processing speed. Since a transport connection, e.g. UDP, should only used by one thread at a time, we need to maximize the time spent handling this connection, e.g. reading and writing packets. Serializing and deserializing can be done by multiple threads in parallel since this is a compute-intense task and does not require any dependents except for read data.

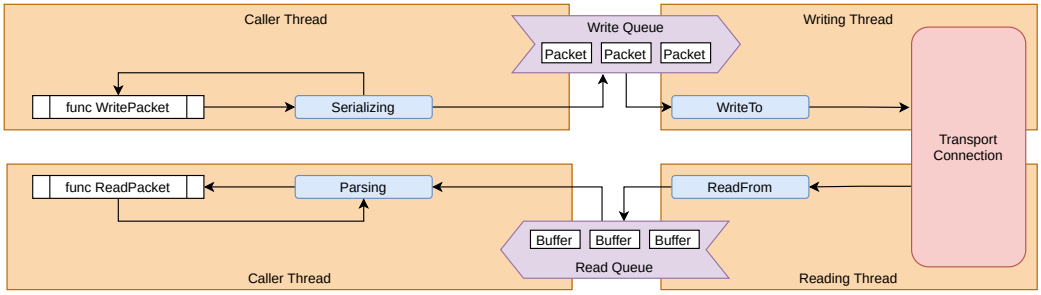

Figure 5.3: Multi-Threaded Packet Processing

We introduce multiple buffered processing queues for parallel packet processing as shown in Figure 5.3, a write queue for serialized packets that are ready to be sent out, and a read queue for read packets that are ready to be parsed and processed from the application. Therefore, we can spawn two independent threads for sending and receiving packets, e.g. writing thread and reading thread. We still use the caller thread for serializing and parsing packets. It also allows to use the read and write methods from multiple threads concurrently to speed up packet processing on the same connection even further.

We tried to implement four different packet processing queues, e.g. different processing queues for serializing, writing, reading and parsing packets, but Go provides an opinionated interface for reading and writing packets. If a packet needs to be read, the reference to an empty packet is passed to this function call. The interface expects the read packet to be written into this reference. Decoupling this into a different processing queue, creating and releasing new packets for every read packet and only copying the contents of the parsed packet leads to many allocations which are dropped shortly after allocation, following the garbage collector slows down the entire application due to this workload. Therefore, we dropped this idea.

5.3 Own SCION Connection Implementation

The implementation of a multi-threaded SCION connection to solve a performance problem improves bandwidth up to a certain threshold. Even though the available resources can be better utilized by multi-threading, no further enhancement are observable if threads are scaled up more. Therefore, we decided to implement our own SCION connection to solve the problem of *Slow Packet Handling* again.

This implementation makes two basic assumptions. The receiver specified when establishing the connection is fixed for the entire connection lifetime. Furthermore, we do not expose the information which is contained in the SCION headers. This leads to the fact that in our implementation in Go, only `net.Conn` is implemented as an interface, not `net.PacketConn`, as is usual, for example, with the existing SCION connection or UDP connections [26][18].

In usual circumstances a SCION connection is established between two clients and both clients will only communicate with each other utilizing this connection. Therefore, we drop the possibility to send SCION packets to arbitrary SCION addresses. This allows more caching possibilities and removes complexity.

We also reduce the nesting of connection implementations. Currently, there is an connection implementation that parses the full SCION packet from the underlying transport connection wrapped into another connection implementation extracting the payload from the SCION packet [26]. This design allows more SCION configuration possibilities which are not needed in the usual circumstances but in SCION core components. Our connection implementation will not expose the SCION packet to the user resulting in providing an interface to transmit data hiding SCION implementation details.

Avoiding the implementation of `net.PacketConn` saves some steps and keeps the implementation clear for a proof of concept. However, a complete implementation of `net.PacketConn` is possible without problems and would also remove the restriction of the fixed receiver, since the interface specifies that packets can be sent to abitrary receivers. This would make this optimized SCION connection drop-in compatible with the previous implementation.

Previously, when the user wanted to send the entire packet, it was completely re-serialized from scratch [26]. We have defined that the receiver will not change over the connection lifetime neither does the SCION path. This means that most parts of the package remain identical and only the `PayloadLen` and some parts of the L4 payload change. The SCION packet serialization process, implemented utilizing `gopacket`, degrades performance drastically for reasons unknown to us. Therefore, we serialize an empty packet on connection creation using the fixed sender and receiver addresses. The resulting buffer is used as a template for all sent packets. If the user wants to send an arbitrary payload, we only need to append the L4 payload to the buffer and change the `PayloadLen` field within the SCION packet. It is also reusable over the connection lifetime.

The reference connection implementation parses the entire packet from the

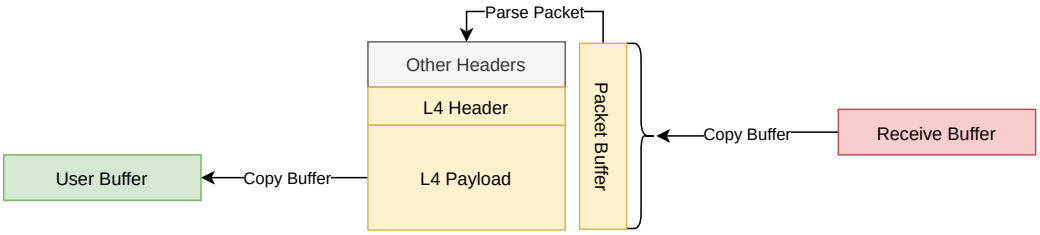

Figure 5.4: Current Incoming Packet Parsing Process

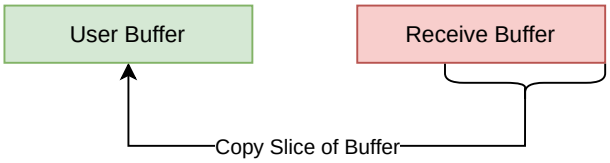

Figure 5.5: Proposed Incoming Packet Parsing Process

underlying transport connection as shown in Figure 5.4 [26]. However, if this connection is used outside of SCION core components in a user application, then most often only the payload is extracted and the rest discarded. Therefore, we can reduce this slow, memory-intensive two-step process into a one-step process only parsing the payload as shown in Figure 5.5. Consequently, we can parse the `PayloadLen` from a fixed position in the SCION header (seen in Figure A.1) and extract `PayloadLen` bytes from the end of the packet without utilizing the parsing method from `gopacket` which is rather slow for the complex SCION packet.

Since there have been many attempts to optimize the SCION connection, it is unsure how much a new implementation will bring in terms of performance improvement [8]. However, a major advantage of a new, more straight-forward implementation is the readability and better maintainability. The proposed implementation has under 500 lines of code, which only partially rely on existing code in `snet` [27].

6. Evaluation

This chapter describes existent bandwidth measurements from Gartner, Christensen and Richter and Syzov et al. and explains the benchmark setup. Afterwards, it evaluates the design ideas, described in Chapter 5, i.e. bypassing the Dispatcher, implementing a multithreaded SCION connection and implementing an optimized SCION connection, as multiple candidates within different scenarios, i.e. a scientific and a real-world scenario. At the end, all results are summarized and related to each other.

6.1 Existing Measurements

Gartner has already made some measurements in a similar environment in their work "Improving SCION Bittorrent with efficient Multipath Usage", which served a different purpose, but can be used as a guide. There, as already mentioned in Related Work, speeds of up to 600 MBit/s could be measured [11].

Christensen and Richter has evaluated UDP performance in their work "Achieving reliable UDP transmission at 10 Gb/s using BSD socket for data acquisition systems", completely independent of SCION. They were able to achieve a throughput of 8 GBit/s over a single connection in an environment of a similar performance level as our test environment presented later, especially a fixed MTU of 1472 [7]. They used the C++ interfaces from the direct kernel functions and thus had no overhead from an abstracting programming language such as Go.

Syzov et al. have measured similar results in their work "Custom udp-based transport protocol implementation over DPDK" [28]. They were able to achieve a transmit speed of 10 GBit/s and a receive speed of almost 7 GBit/s, also on a UDP connection. These results were also measured in a similar environment.

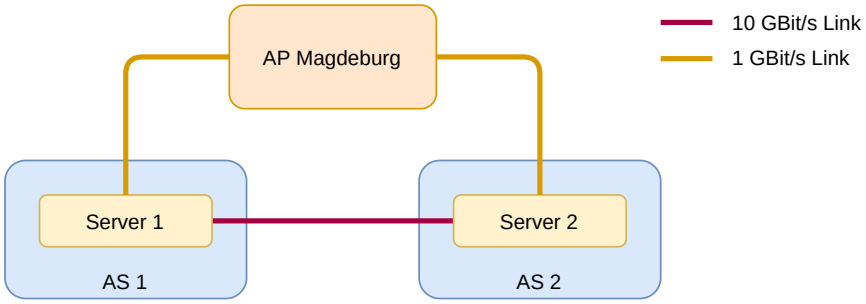

Figure 6.1: Evaluation Topology (Between ASes)

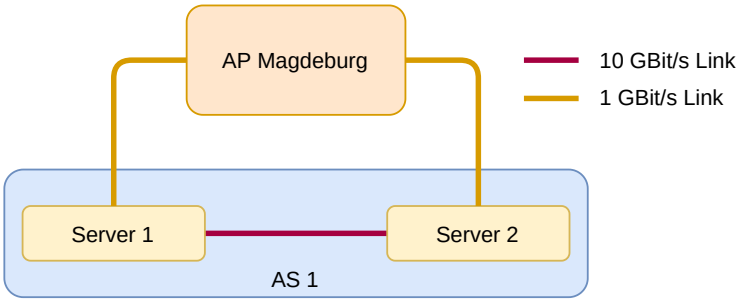

Figure 6.2: Evaluation Topology (Same AS)

6.2 Setup

We use in our experiments the same testbed as Gartner [11]. It features two dedicated server and a 10 GBit/s point-to-point connection to reduce noise and errors.

6.2.1 Topology

On this testbed, we use two different topologies within our experiments. In each topology both servers are attached to an attachment point of SCIONLab[15] using an 1GBit/s link. Additionally, there exists a 10 GBit/s link between both servers. All bandwidth measurements are done by utilizing the 10 GBit/s link. Each link has a fixed MTU of 1500 bytes.

Our two topologies differ in their use of ASes. *Synthetic Data Transmit Between ASes*, *File Transfer via QUIC*, and *File Transfer via PARTS* uses two ASes with one server per AS, as shown within Figure 6.1. *Synthetic Data Transmit Within AS* uses both servers within the same AS, as shown within Figure 6.2.

6.2.2 Machines

In our testbed, we use two similar machines and a direct 10 GBit/s connection. A machine contains an Intel Xeon Silver 4114 CPU with 10x2.20GHz, 48GB of DDR4 RAM with a clock rate of 2666MHz, an 1 GBit/s network interface, and an 10 GBit/s network interface. Each machine runs a copy of Ubuntu 20.04.1 LTS with the latest patches and the SCIONLab December 2020 Release. We also added a High Speed Border Router (HSR) from Anapaya Systems AG to eliminate known restrictions within the open-source Border Router [26][1]. Otherwise, the SCION installation is as adviced by default.

6.2.3 Scenarios

We evaluated two different scenarios within this work. In our first scenario, *Synthetic Data Transmit*, we transmit and read synthetic data as fast as possible to establich a ground truth of the possible performance. In our second scenario, *File Transfer*, we aim for a more real-world scenario where we transfer a file from one server to another.

Synthethic Data Transmit is realized by a two applications: a server application and a client application. The server applications receives packets sent by the client application. Both applications process as many packets as they can with no rate limiting. Both applications utilize one to many connections in parallel to overcome possible limitations of single connections. We collect data about the sending goodput and throughput, receving goodput and throughput and packet loss.

File Transfer is done by utilizing a reliable transport protocol, i.e. QUIC and PARTS to add reliable transport to SCION, since it suffers from the missing packet order and no guarantees of packet delivery without an additional overlay protocol [22]. The server application receives a file sent by the client application using the overlay protocol. Also, both applications utilize one to many connections in parallel, too. We measure the time for the complete filetransfer including possible retransmissions and other protocol-specific actions and check the correctness of the received file afterwards. Since we measure such small packet loss in our *Synthetic Data Transmit*, we do not measure it within *File Transfer*, because we do not assume changes in packet loss for sending with lower than maximum bandwidth.

6.2.4 Candidates

We use different candidates in both scenarios. We measure each experiment once with no changes to the existing codebase as **Dispatcher**. Additionally, we measure each experiment with the needed changes to bypass the Dispatcher, which have been described in 5.1, named **xiondp**. Since we discovered and solved further performance bottlenecks by utilizing multi-threading, we also measure all experiments with the optimized multi-threading version of the existing SCION connection, introduced in 5.2, as **xiondp Multi-Thread**. Last but not least, we also include our own implementation of a SCION connection, outlined in 5.3, in the evaluation as **xiondp Optimized**.

We also considere some other candidates, but we dropped them for various reasons. We did not evaluate any other connection implementations not bypassing the Dispatcher, even if that would have been technically possible. We saw the limitation within the Dispatcher within our first tests that cannot be fixed by utilizing another connection implementation. We also have not improved the xiondp Optimized connection further since we already reached maximum possible performance in our testbed and could not get better results in any way.

6.3 Experiments

6.3.1 Synthetic Data Transmit Between ASes

We used this basic experiment to test our design approaches described in Chapter 5 to solve our performance bottlenecks described in Chapter 4. We ran the *Synthetic Data Transmit* scenario between two distinct ASes with 1 to 12 parallel SCION connections.

Figure 6.3 shows, that the **Dispatcher** candidate was able to send 1.2 GBit/s goodput. These measurements are slightly higher than the expected results shown in previous experiments by Gartner [11]. However, optimal conditions prevail in our experiment, whereas in the existing experiments the entire BitTorrent overhead had to be handled as well.

The **xiondp** candidate shows an increase to a maximum of 4.6 GBit/s in sending throughput. It was also capable of receiving packets with the same speed. At this point we found that the potential bottleneck is no longer outside the application, as before in the Dispatcher, but is now part of the application, probably within the connection implementation.

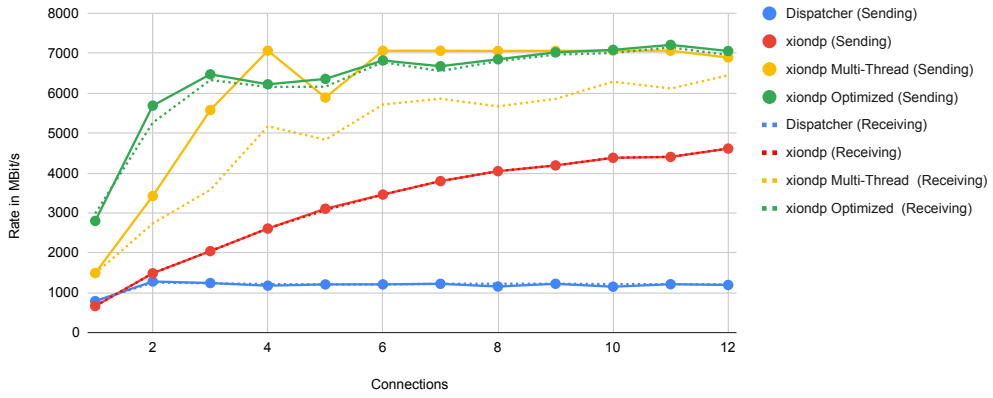

Figure 6.3: Sending and Receiving Rate of Synthetic Data Transmit Between ASes (generated from Table A.1 and Table A.3)

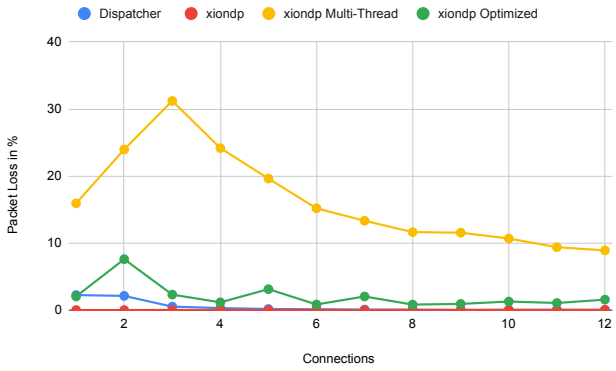

Figure 6.4: Packet Loss of Synthetic Data Transmit Between ASes (generated from Table A.5)

The next candidate **xiondp Multi-Thread** shows a great performance improvement compared to the last candidate. It was able to send up to 6.8 GBit/s but had troubles receving the sent packets in time. We assume a performance limitation in the packet decoding and serializing process which causes the packets that are not processed fast enough to be dropped. There is also a decreased sending performance at 5 connections, which occurs reproducibly on our test environment. We will analyze this in the future.

The candidate **xiondp Optimized** does not improve the maximum transmission speed significantly compared to the multi-threaded approach, but it was able to receive packets as fast as they were sent. This makes it possible to use it in real-world scenarios. It also doubles the single-connection performance, which is especially important for non-scientific applications.

Figure 6.4 shows, that we experience no major packet loss within this experiment except for **xiondp Multi-Thread** candidate. As noted earlier, the approach seems to have problems reading the packets as fast as they arrive, presumably by decoding them too slowly. We handle each connection with separate threads in this candidate, which means that the number of threads used scales linearly with the number of connections. We see a sharp increase in packet loss until the connection throughput is fully saturated. This packet loss happens because the sender with more threads performs better than the receiver. After this peak in packet loss, it decreases as more connections are used, since the sender now divides the same number of packets, since the connection throughput is saturated, among more connections. These connections are processed with more threads on the receiver side and thus allow better receive performance. However, notable packet loss can still be measured, which disqualifies this candidate for any real-world use. Figure 6.4 also shows low packet loss of the **xiondp Optimized** candidate, which is not caused by the connection candidate itself. Presumably, the border router in this particular configuration is already the problem here, which will be further evaluated in the next experiment.

Throughout the experiment, we saw that more connections could significantly increase throughput once the initial Dispatcher bottleneck was resolved. Furthermore, we could not achieve more than a goodput of 7 GBit/s with any implementation. Even though this corresponds to a throughput of 7.5 GBit/s, this should not be the limit of the potential speed, since there were still enough resources available in our testbed. We have also improved single-connection performance, which is very important for non-scientific applications. With a goodput of almost 3 GBit/s, this performance is already much better than the previous one, which could only reach 800 MBit/s.

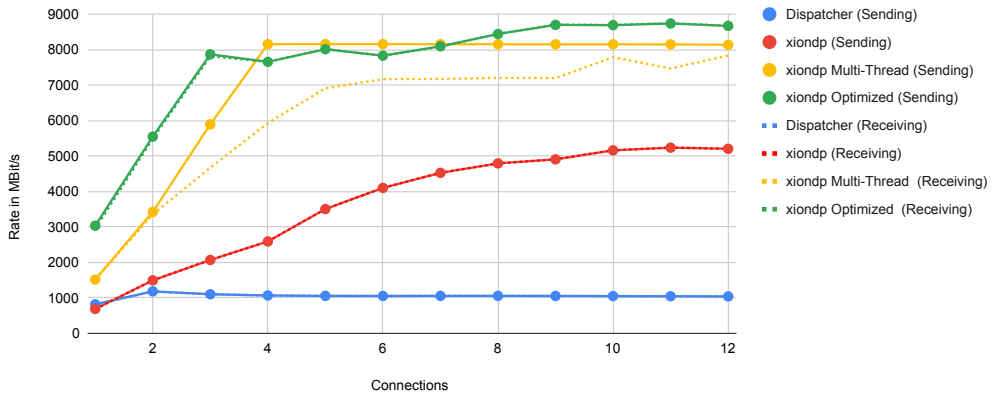

Figure 6.5: Sending and Receiving Rate of Synthetic Data Transmit Within AS (generated from Table A.6 and Table A.8)

Our candidates show greatly improved goodput compared to the Dispatcher for synthetic data. We see performance improvements for all different candidates up to factor 6. Single-connection performance in particular could also be quadrupled with the **xiondp Optimized** candidate.

6.3.2 Synthetic Data Transmit Within AS

Since we did not know the reason for our 7 GBit/s limitation and still aim to saturate the 10 GBit/s link, we decided to repeat the last experiment and eliminate the HSR as a potential cause. Since a Border Router is crucial to transmit data between two different ASes, both servers are configured to be part of the same AS for this experiment. This changes the path of the data slightly so that no Border Router is used to communicate between the two servers. We repeated the same *Synthetic Data Transmit* scenario with 1 to 12 parallel SCION connections.

Figure 6.5 shows the measured sending and receiving goodput in this experiment. Although we do not see any significant differences compared to the previous experiment for small numbers of connections, the 7 GBit/s limit seems to disappear and even slower candidates like **xiondp** were able to achieve better results. **xiondp Multi-Thread** and **xiondp Optimized** were able to send 8.1 GBit/s and 8.6 GBit/s over multiple SCION connections. There are no changes for the results of the **Dispatcher** candidate. Since 8.6 GBit/s goodput corresponds to 9.4 GBit/s throughput, we reached the limitations of our testbed.

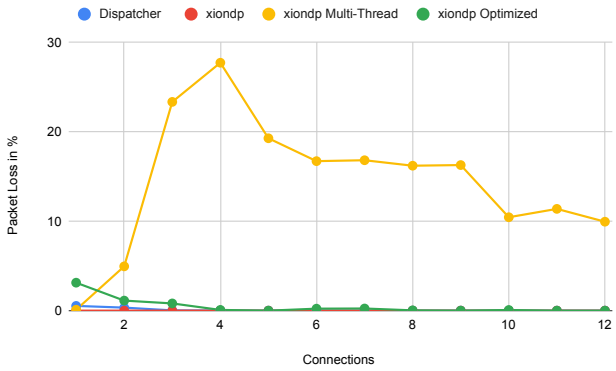

Figure 6.6: Packet Loss of Synthetic Data Transmit Within AS (generated from Table A.10)

In our previous experiment, the sending and receiving bandwidth did not scale linearly. It would be expected utilizing multiple parallel independent connections. Without an active Border Router, we achieve almost a linear bandwidth scaling until hitting our testbed limitations. We also see a lot less packet loss for fast connections, e.g. **xiondp Optimized**, that was probably previously introduced by the Border Router.

In short, this shows another performance improvement and raises the maximum possible goodput to 8.6 GBit/s. It is probably not that important for non-scientific measurements since the performance improvement is only about 15% compared to the same experiment with a Border Router. We assume the combination of application and Border Router on the same host leads into potential bottlenecks, so in Future Work we aim to verify this assumption with testing in setups where this is not the case.

6.3.3 File Transfer via QUIC

Next, we present a real-world experiment to make sure our changes also work for non-scientific applications. Therefore, we ran the *File Transfer* scenario with QUIC as transport overlay protocol from 1 to 12 parallel connections between two distinct ASes. Not all candidates used in the previous experiments are suitable for the real-world experiment. Due to the high packet loss of our **xiondp Multi-Thread** candidate, as already mentioned within 6.3.1, we dropped this candidate for this real-world experiment. We also dropped the **xiondp Optimized** candidate for implementation-specific reasons, since quic-go requires a `net.PacketConn` implementation that the optimized SCION

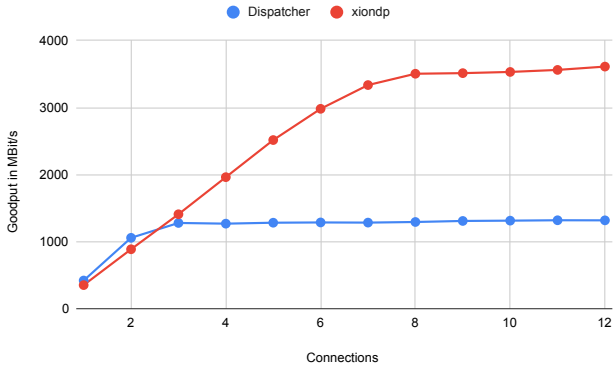

Figure 6.7: Goodput of File Transfer via QUIC (generated from Table A.11)

connection cannot provide as stated in 5.3 [16].

Figure 6.7 shows the goodput for the two remaining candidates. The **Dispatcher** candidate performs as expected transmitting the file at around 1.3 GBit/s goodput. The candidate **xiondp** outperforms the other candidate with 4 or more connections and we were able to transmit a file with 3.6 GBit/s goodput. Although a goodput of 4.5 GBit/s could be achieved in the synthetic experiment, the results are more than satisfactory considering the additional protocol and compute overhead.

In summary, **xiondp** shows, that our Dispatcher Bypass is also applicable in real-world scenarios and outperforms existing approaches by more than a factor of 3.

6.3.4 File Transfer via PARTS

Since QUIC and its go implementation `quic-go` do not work very well for our use cases and are not the optimal choice for a large file transmit, we decided to run the same *File Transfer* scenario with PARTS as transport overlay protocol. Since Parts is in an early stage of development, 12 connections are not currently supported. Therefore, we use 1 to 8 parallel connection between two distinct ASes for each proposed candidate but **xiondp Multi-Thread**. We had the same problem as described within the previous section and decided to remove the candidate due to the high packet loss. Fortunately, with the PARTS implementation, we can evaluate the **xiondp Optimized** candidate.

Figure 6.8 shows the goodput for our candidates. As always, the **Dispatcher** candidate performs as expected. It is able to transfer the file with a good-

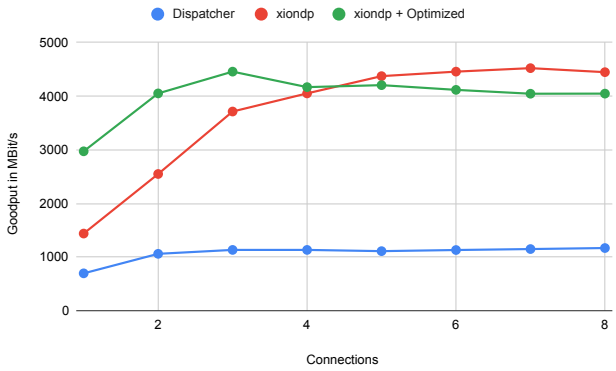

Figure 6.8: Goodput of File Transfer via PARTS (generated from Table A.12)

put of 1.2 GBit/s. Although the transfer rate is lower than in a previous experiment, it is within the expected range, considering that PARTS also requires processing power that is no longer available for sending and receiving packets. The **xiondp** candidate performs slightly better as expected from the previous experiment and is able to transfer the file at a goodput of 4.5 GBit/s. Also the new **xiondp Optimized** candidate performs similar to the previous candidate and is able to achieve a goodput of 4.4 GBit/s utilizing only three connections.

Since the **xiondp Optimized** candidate performed very well with synthetic data, we actually expected it to outperform the **xiondp** candidate. The failure to meet this expectation may probably be a problem with the PARTS protocol not utilizing all the available bandwidth. Fortunatly, it works very well in single connection applications and it is able to transmit a file with one connection at about 3 GBit/s goodput. It doubles the possible goodput of the **xiondp** candidate.

In summary, **xiondp** and **xiondp Optimized** show that real-world applications can profit from raised goodputs. We were able to outperform the existing **Dispatcher** approach by a little more than factor of 3.

6.4 Summary

The experiments evaluate the performance of our proposed changes within Chapter 5 in two different scenarios, i.e. synthetic data and file transfer. The candidates are a reasonable combination of the various optimizations as they could be used in real-word scenarios.

There are two impactful performance bottlenecks in the current SCION reference implementation, i.e. the Dispatcher and the performance of the client library. We showed, that our changes can overcome both bottlenecks in scientific and real-world applications.

In the first two experiments it was tested whether and to what extent the proposed changes cause a change in the transmission speed. For this purpose, synthetic data packets were transferred between two servers as fast as possible. From this, the maximum sending and receiving rate were determined, and it was also recorded how many packets were lost. Initially, only four times as many packets could be transmitted with candidate **xiondp** as with the reference implementation. Six times as many packets could be transmitted utilzing the **xiondp Multi-Thread** and **xiondp Optimized** candidate. In addition, the HSR setup was identified as another possible bottleneck. If the HSR is removed from the test setup, then the goodput can be increased sevenfold.

In the other two experiments it was tested whether the proposed optimizations also work in real-world scenarios. For this purpose, the speed of a file transfer between two servers via different overlay protocols was tested. The file transfer was accelerated with the candidates **xiondp** and **xiondp Optimized** by at least a factor of 3 compared to the reference implementation. Additionally, one candidate was declared as unusable in real-world applications due to a high packet loss rate.

We were able to solve both major performance bottlenecks and show that they are applicable in almost all SCION applications. With a bypass of the Dispatcher, initial throughput improvements of up to a factor of 3 are possible; with further optimization of the SCION connection implementations, it was also shown that the throughput can be improved by factor 4 compared to the reference implementation.

7. Conclusion

The objective of this thesis was to answer the following research question: To what extent can modern network-related technologies, especially XDP, accelerate the performance of end-user applications in SCION without changing APIs? To answer this question, two scenarios, a scientific scenario and a real-world scenario, were evaluated with four different candidates representing the different optimizations.

We have solved several bottlenecks with this thesis. The most impactful bottleneck so far was in the Dispatcher, which could be solved with the help of XDP. XDP offers the possibility to perform basic operations on incoming packets and thus to execute the functionality of the Dispatcher relaying packets to the registered applications in the kernel space. With this change alone, we were able to measure almost a quadrupling of goodput during our synthetic experiment from 1.2 GBit/s to 4.6 GBit/s. In our real-world experiment, we were still able to measure a threefold increase in goodput and the goodput from 1.2 GBit/s to 3.6 GBit/s.

However, it was also determined that solving this bottleneck alone would not lead to our goal and now we also need to process packets efficiently and quickly in the client applications. For this purpose, we implemented our own optimized SCION connection which makes assumptions to gain performance outlined in Chapter 5.3 that can be lifted in further research outlined in Chapter 8. Those optimizations increases the goodput once again. Thus, we were able to increase the goodputs from 1.2 GBit/s to 8.6 GBit/s and so sevenfold the performance in synthetic experiments.

In real-world applications, goodput could only be more than tripled from 1.2 GBit/s to 4.5 GBit/s. This is probably because the overlying protocols, i.e. QUIC and PARTS, are not or not yet optimized to transmit such high data rates. However, we were also able to almost quadruple the single-connection goodput, i.e. the goodput that can be achieved with only a single connection. This makes it easier to implement applications that rely on fast network speeds, because they can now save the overhead of connection management.

In summary, XDP had a major impact here in eliminating the bottlenecks that had previously required many hours of optimization [8]. It alone was able to

quadruple the goodput. However, in comparison to bare-metal perforamnce, this is not sufficient, but in this context it is an important foundation for further optimizations. The libraries within the applications, which send and receive the packets, must also be built with similar performance in mind, so that this advantage may be exploited. Further performance advantages, i.e. optimized SCION connection, can be used with certain restrictions, but this is purely optional.

8. Future Work

At the end, there are always unanswered questions that offer opportunities to conduct further research. Besides possibilities for further optimization, the reasons for the bottlenecks can also be explored.

For a network protocol that aims to be a considerable alternative to the existing Internet [22], there should be unit tests and performance benchmarks for such critical components as the SCION connection, which is used by all parts of the reference implementation. This is not the case so far and leads to a certain hurdle to touch and optimize existing code. In our work, missbehaviour of our connection implementations occurred, which subsequently led to erroneous measurements. This would have been noticed with extensive tests.

In our implementation of the optimized SCION connection, we discovered two limitations compared to the normal SCION connection, namely dropping information of the origin and packets may only be sent to a fixed remote destination. This can be solved with the implementation of `net.PacketConn`, making this optimized SCION connection fully drop-in compatible with the existing SCION connection.

In the evaluation, we discovered an unexpected limitation by running the HSR and the application on the same machine. This may be a problem with simultaneous reading and writing on the same network interface of the application and HSR, however we have not investigated this further, as this would have gone beyond the scope of this work. However, it would be a relevant further research topic to increase the capacity of the SCION network.

In addition, many new possibilities arise when XDP can be used. The bypass of the Dispatcher was a concise example of the implementation of such problems with XDP. In a further step, a Border Router or similar could also be accelerated with the help of XDP, even if this would become more complex due to the required data. The reference implementation here could also benefit from the possible speed improvements that can be achieved by accelerating the critical path with XDP. For example, only the forwarding of SCION data packets could be accelerated via XDP and all other SCION packets could be handled as usual, as we did within xiondp.

Furthermore, the entire SCION connection used in the client applications

could be converted to a UDP connection using XDP. This would imply that applications could use the SCION network like a normal UDP connection and with the help of XDP all connections would be wrapped transparently into a SCION connection when sending and unpacked from a SCION connection when receiving. The receive direction is trivial to implement, the send direction requires more effort because the addressing of SCION and UDP is fundamentally different.

In summary, there is much work to be done both within the reference implementation and in a potential new high-performance implementation. Different companies maintain their own SCION implementation. Often the reference implementation lags behind the closed source implementations performance-wise. This should be addressed, also to drive the development of applications using SCION, by improving the speed of the reference implementation. Many parts of the data path of the reference implementation can be accelerated using modern technologies, e.g. XDP. If this is still not desired within the reference implementation [25], it is possible to develop a high-performance implementation based on the reference implementation. Some components such as the Border Router could benefit greatly from this.

Literature

[1] Anapaya. *Next-Generation Internet*. URL: *https://www.anapaya.net/*.

[2] Gilberto Bertin. "XDP in practice: integrating XDP into our DDoS mitigation pipeline". In: *Technical Conference on Linux Networking, Netdev*. Vol. 2. 2017.

[3] Shourya P Bhattacharya and Varsha Apte. "A measurement study of the Linux TCP/IP stack performance and scalability on SMP systems". In: *2006 1st International Conference on Communication Systems Software & Middleware*. IEEE. 2006, pp. 1–10.

[4] *BPF and XDP Reference Guide - cilium 1.10.90 documentation*. URL: *https://docs.cilium.io/en/latest/bpf/*.

[5] *BPF Documentation*. URL: *https://www.kernel.org/doc/html/latest/bpf/index.html*.

[6] *BPF Kernel Versions - iovisor/bcc*. May 2021. URL: *https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md*.

[7] Morten Jagd Christensen and Tobias Richter. "Achieving reliable UDP transmission at 10 Gb/s using BSD socket for data acquisition systems". In: *Journal of Instrumentation* 15.09 (2020), T09005.

[8] Sergiu Costea. *dispatcher: Replace with SCMP Daemon · Issue #3961 · scionproto/scion*. Jan. 2021. URL: *https://github.com/scionproto/scion/issues/3961#issue-779233886*.

[9] Jan Engelhardt. *Packet flow in Netfilter and General Networking*. File: `Netfilter-packet-flow.svg`. 2019. URL: *https://en.wikipedia.org/wiki/File:Netfilter-packet-flow.svg*.

[10] Matthias Frei and François Wirz. *Hercules - Bulk Data Transfer over SCION*. Tech. rep. ETH Zurich. URL: *https://www.scion-architecture.net/pages/scion_day/slides/Hercules%20-%20Bulk%20Data%20Transfer%20over%20SCION.pdf*.

[11] Marten Gartner. "Improving SCION Bittorrent with efficient Multipath Usage". PhD thesis. 2020.

[12] Oliver Hohlfeld et al. “Demystifying the Performance of XDP BPF”. In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2019, pp. 208–212.

[13] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. RFC Editor, May 2021.

[14] Martin Koppehel. *Case Study of BitTorrent over SCION*. URL: *https://mko.dev/research/scion-bittorrent.pdf*.

[15] Jonghoon Kwon et al. “SCIONLAB: A Next-Generation Internet Testbed”. In: *2020 IEEE 28th International Conference on Network Protocols (ICNP)*. IEEE. 2020, pp. 1–12.

[16] Lucas-Clemente. *lucas-clemente/quic-go*. URL: *https://github.com/lucas-clemente/quic-go*.

[17] Sebastiano Miano et al. “Creating complex network services with ebpf: Experience and lessons learned”. In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE. 2018, pp. 1–8.

[18] *net · pkg.go.dev*. URL: *https://pkg.go.dev/net*.

[19] netsys-labs. *netsys-labs/parts*. URL: *https://github.com/netsys-labs/parts*.

[20] ETH Zurich Network Security Group. URL: *https://www.scion-architecture.net/*.

[21] Cédric Neukom. “High-Performance File Transfer in SCION”. MA thesis. ETH Zurich, 2020.

[22] Adrian Perrig et al. *The SCION Architecture*. Cham: Springer International Publishing, 2017, pp. 17–42. ISBN: 978-3-319-67080-5. DOI: *10.1007/978-3-319-67080-5_2*. URL: *https://doi.org/10.1007/978-3-319-67080-5_2*.

[23] Dominik Scholz et al. “Performance implications of packet filtering with Linux eBPF”. In: *2018 30th International Teletraffic Congress (ITC 30)*. Vol. 1. IEEE. 2018, pp. 209–217.

[24] *SCION Project Documentation*. URL: *https://scion.docs.anapaya.net/*.

[25] Scionproto. *Changes to Dispatcher · Issue #4059 · scionproto/scion*. URL: *https://github.com/scionproto/scion/issues/4059*.

[26] *scionproto/scion*. URL: *https://github.com/scionproto/scion*.

[27] *snet package - github.com/scionproto/scion/go/lib/snet*. URL: *https://pkg.go.dev/github.com/scionproto/scion/go/lib/snet*.

[28] Dmytro Syzov et al. “Custom udp-based transport protocol implementation over DPDK”. In: *Proceedings of International Conference on Applied Innovation in IT*. Vol. 7. 1. Anhalt University of Applied Sciences. 2019, pp. 13–17.

[29] Marcos Vieira et al. “Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications”. In: *ACM Computing Surveys (CSUR)* 53 (Feb. 2020), pp. 1–36. DOI: *10.1145/3371038*.

[30] Xin Zhang et al. “SCION: Scalability, control, and isolation on next-generation networks”. In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 212–227.

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning. I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, September 7, 2021

Signature

List of Figures

List of Tables

List of Abbreviations

TCP	Transmission Control Protocol
SSP	SCION Stream Protocol
UDP	User Datagram Protocol
SCION	Scalability, Control, and Isolation on Next-Generation Networks
BGP	Border Gateway Protocol
NIC	Network Interface Controller
eBPF	Extended Berkeley Packet Filter
XDP	Express Data Path
API	Application Programming Interface
ISD	Isolation Domain
AS	Autonomous System
RAINS	RAINS, Another Internet Naming Service
SCMP	SCION Control Message Protocol
ICMP	Internet Control Message Protocol
SCION/UDP	User Datagram Protocol over SCION
SCION/TCP	Transmission Control Protocol over SCION
QUIC	Quick UDP Internet Connections
PARTS	Path-Aware Reliable Transport over SCION
DPDK	Data Plane Development Kit
BSD	Berkeley Software Distribution

CPU	Central Processing Unit
MTU	Maximum Transmission Unit
IP	Internet Protocol
DDoS	Distributed Denial of Service
HSR	High Speed Border Router

A. Appendix

Design

SCION Packet Structure

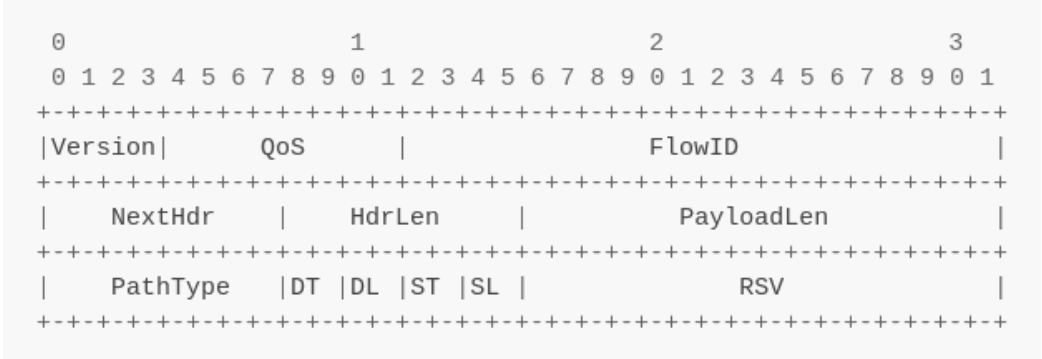

Figure A.1: SCION Common Header [22]

Evaluation

Synthetic Data Transmit (SDT) Between ASes

In this chapter, the measurement results of the experiment *Synthetic Data Transmit Between ASes* are tabulated. The graphs of the evaluation chapter are based on these data.

Table A.1: SDT Between AS - Sending Goodput (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	787	662	1489	2797
2	1279	1485	3422	5688
3	1240	2040	5577	6473
4	1174	2606	7066	6223
5	1206	3104	5890	6358

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
6	1207	3458	7063	6820
7	1221	3795	7064	6676
8	1153	4045	7059	6850
9	1221	4189	7061	7023
10	1148	4380	7066	7086
11	1211	4402	7065	7211
12	1192	4611	6894	7057

Table A.2: SDT Between AS - Sending Throughput (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	848	713	1605	3014
2	1378	1600	3688	6129
3	1336	2198	6010	6975
4	1265	2808	7614	6706
5	1300	3344	6347	6852
6	1301	3726	7611	7350
7	1316	4090	7612	7194
8	1242	4359	7607	7382
9	1316	4514	7609	7568
10	1237	4720	7614	7635
11	1305	4743	7614	7770
12	1284	4969	7429	7604

Table A.3: SDT Between AS - Receive Goodput (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	789	663	1478	2983
2	1252	1485	2733	5260
3	1234	2041	3578	6330
4	1214	2609	5172	6156
5	1205	3068	4833	6163
6	1206	3458	5716	6777
7	1221	3797	5861	6555
8	1223	4046	5669	6803
9	1222	4189	5857	6966

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
10	1214	4382	6288	7008
11	1211	4404	6117	7147
12	1206	4613	6448	6958

Table A.4: SDT Between AS - Receive Throughput (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	850	715	1592	3214
2	1350	1600	2945	5668
3	1330	2199	3856	6822
4	1308	2812	5573	6634
5	1298	3306	5208	6641
6	1300	3726	6160	7303
7	1315	4092	6316	7063
8	1318	4360	6108	7331
9	1317	4514	6312	7507
10	1308	4722	6776	7552
11	1305	4745	6592	7702
12	1300	4971	6948	7498

Table A.5: SDT Between AS - Packet Loss (in %)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	2.24182	0	15.929	2.032
2	2.11663	0	23.968	7.5935
3	0.51305999	0.0186	31.204	2.3071
4	0.29898	0.0112	24.172	1.1559
5	0.16294	0	19.622	3.1209
6	0.12926	0.0105	15.181	0.8363
7	0.07474999	0.0062	13.326	2.0306
8	0.05952	0.0322	11.633	0.819
9	0.02549	0.0259	11.547	0.9216
10	0.04864999	0.023	10.678	1.2763
11	0.0519	0.0213	9.3842	1.057
12	0.01404999	0.0456	8.9004	1.5561

Synthetic Data Transmit (SDT) Within AS

In this chapter, the measurement results of the experiment *Synthetic Data Transmit Within AS* are tabulated. The graphs of the evaluation chapter are based on these data.

Table A.6: SDT Within AS - Sending Goodput (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	818	686	1516	3034
2	1184	1494	3423	5549
3	1102	2069	5894	7869
4	1068	2591	8154	7657
5	1056	3505	8156	8013
6	1053	4101	8155	7831
7	1056	4526	8154	8095
8	1056	4793	8153	8442
9	1054	4906	8151	8700
10	1050	5161	8152	8692
11	1046	5237	8149	8739
12	1041	5204	8138	8671

Table A.7: SDT Within AS - Sending Throughput (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	881	739	1634	3269
2	1276	1610	3688	5980
3	1187	2229	6351	8480
4	1151	2792	8786	8252
5	1138	3777	8788	8635
6	1134	4419	8788	8439
7	1138	4877	8786	8723
8	1138	5165	8785	9097
9	1136	5287	8783	9374
10	1132	5562	8784	9367
11	1127	5643	8782	9417
12	1122	5608	8770	9344

Table A.8: SDT Within AS - Receive Goodput (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	814	686	1517	2940
2	1180	1494	3353	5490
3	1102	2069	4671	7813
4	1068	2592	5931	7657
5	1056	3506	6910	8019
6	1053	4102	7169	7826
7	1056	4528	7174	8084
8	1056	4795	7204	8449
9	1054	4909	7199	8708
10	1050	5163	7787	8695
11	1046	5240	7472	8749
12	1041	5207	7832	8675

Table A.9: SDT Within AS - Receive Throughput (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	877	739	1635	3168
2	1272	1610	3613	5916
3	1187	2230	5033	8419
4	1151	2793	6391	8251
5	1138	3778	7446	8641
6	1134	4420	7726	8433
7	1138	4880	7731	8712
8	1138	5167	7763	9104
9	1136	5290	7758	9384
10	1132	5564	8391	9370
11	1127	5646	8051	9428
12	1122	5611	8439	9348

Table A.10: SDT Within AS - Packet Loss (in %)

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
1	0.53269	0.00011	0.10505	3.1311
2	0.34005	0.00172	4.95776	1.1261
3	0.03019	0	23.3618	0.8092

Connections	Dispatcher	xiondp	xiondp Multi-Thread	xiondp Optimized
4	0.0167	0.00332	27.744	0.0828
5	0.00204	0	19.2923	0
6	0.00092	0.00029	16.7311	0.2237
7	0.00055	0	16.8329	0.2419
8	0.00042	0	16.2206	0.0345
9	0.00324	0.00075	16.2933	0.0034
10	0.00036	0	10.4542	0.0743
11	0	0	11.3893	0
12	0	0	9.95443	0

File Transfer via QUIC

In this chapter, the measurement results of the experiment *File Transfer via QUIC* are tabulated. The graphs of the evaluation chapter are based on these data.

Table A.11: File Transfer via QUIC - File Transmission Speed (in MBit/s)

Connections	Dispatcher	xiondp
1	419.894	351.7317
2	1056.895	886.3849
3	1278.817	1408.477
4	1268.274	1962.021
5	1282.371	2515.808
6	1286.342	2981.765
7	1283.956	3336.224
8	1293.150	3505.147
9	1308.632	3514.043
10	1313.183	3531.972
11	1319.021	3562.264
12	1317.766	3611.826

File Transfer via PARTS

In this chapter, the measurement results of the experiment *File Transfer via PARTS* are tabulated. The graphs of the evaluation chapter are based on these data.

Table A.12: File Transfer via PARTS - Transmission Speed (in MBit/s)

Connections	Dispatcher	xiondp	xiondp Optimized
1	696	1438	2972
2	1060	2549	4052
3	1133	3714	4458
4	1133	4052	4169
5	1110	4374	4206
6	1131	4458	4118
7	1149	4523	4046
8	1168	4448	4048