UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS AND
INFORMATICS

Computer Science Specialization

Bachelor's Thesis

COMMUNICATION FLOW SCHEDULING IN A WEB SERVER

Graduate
Gabroveanu Răzvan Petru

Scientific Supervisor
Olaru Vlad

Bucharest, June 2025

Rezumat

Această lucrare explorează potențialul teoretic al planificării la nivel de flux în serverele web, concentrându-se asupra modului în care strategiile de prioritizare a cererilor pot influența performanța în scenarii care implică multiple cereri HTTP/1.1 concurente pentru conținut static. Această abordare oferă o oportunitate de a experimenta noi forme de control al fluxului de livrare la nivelul aplicației.

În acest scop, este utilizat Stream Control Transmission Protocol (SCTP) ca alternativă la protocolul de transport tradițional TCP. Funcționalitatea integrată de multistreaming a SCTP permite existența mai multor fluxuri independente într-o singură conexiune, oferind un control detaliat asupra ordinii și paralelismului răspunsurilor. Prin contrast, utilizarea pe scară largă a TCP a dus la osificarea protocolului, limitând posibilitatea de a experimenta noi mecanisme de livrare fără a compromite compatibilitatea cu infrastructura existentă.

Sistemul proiectat și implementat în cadrul acestei lucrări constă într-un server HTTP/1.1 minimalist pentru livrarea de conținut static, scris în Rust, un proxy TCP-to-SCTP responsabil de caching și prefetching adaptat conținutului HTML, precum și suport pentru aceste funcționalități. Două politici de planificare, Shortest Connection First și Round Robin, sunt introduse și comparate, fiecare distribuind cererile HTTP pe fluxuri SCTP conform unor principii diferite. Proiectul își propune să evalueze logica de planificare în contextul SCTP, oferind perspective asupra modului în care controlul la nivel de flux poate susține viitoare experimente privind protocoalele de transport.

Abstract

This thesis explores the theoretical potential of stream-level scheduling in web servers, focusing on how request prioritization strategies can affect performance in scenarios involving multiple concurrent HTTP/1.1 requests for static content. This poses an opportunity to experiment with new forms of delivery flow control at the application layer.

To support this exploration, the Stream Control Transmission Protocol (SCTP) is employed as an alternative to the traditional TCP transport layer. The built-in multistreaming feature of SCTP enables multiple independent streams within a single connection, allowing fine-grained control over the ordering and parallelism of responses. In contrast, TCP's widespread use has led to protocol ossification, limiting the ability to experiment with delivery mechanisms without breaking existing stacks.

The system designed and implemented for this thesis consists of a minimalist HTTP/1.1 server for static content delivery written in Rust, a TCP-to-SCTP proxy responsible for HTML-aware caching and prefetching, and support for HTML-aware caching and prefetching. Two scheduling policies, Shortest Connection First and Round Robin, are introduced and compared, each distributing HTTP requests across SCTP streams according to different principles. The project aims to evaluate the scheduling logic in the context of SCTP, offering insights into how stream-level control may benefit future transport protocol experimentation.

Contents

Chapter 1

1 Introduction

1.1 Problem definition

Modern web servers are built using advanced software architectures, either multithreaded or event-driven, that allow high performance in serving HTTP requests. However, a less explored dimension in these architectures is the scheduling of HTTP requests based on policies that aim to optimize request handling performance.

While HTTP/1.1 and TCP have been the foundation of the Web for decades, their simplicity comes with architectural constraints. A request typically arrives over a dedicated connection, and the entire handling process is performed sequentially or by a fixed thread, without considering scheduling strategies. This model does not exploit information such as request size or expected processing time, leading to inefficient use of server resources under high concurrency.

An alternative perspective explored involves rethinking how HTTP requests are handled at the transport layer. Instead of relying on traditional TCP streams, which bind each request to a single, sequentially processed connection, this work investigates how multiple concurrent HTTP flows can be scheduled independently using SCTP's multi-streaming feature.

1.2 Motivation

This thesis investigates an alternative approach: applying scheduling policies to HTTP request flows by leveraging communication-level constructs provided by SCTP (Stream Control Transmission Protocol).

SCTP offers built-in support for multi-streaming and message-oriented delivery within a single association, making it a strong candidate for organizing the transmission of multiple HTTP requests [19].

Such capabilities allow a more structured form of communication between client and server, one where multiple requests can be scheduled and sent concurrently over different SCTP streams. This enables the server to apply flow scheduling policies inspired by CPU scheduling (such as Shortest Job First for small static resources), potentially improving latency and throughput without altering the semantics of HTTP/1.1.

A key motivation behind this work is the idea that client-side techniques such as prefetching can be used to identify a collection of resources linked to a main HTML document. These preloaded requests can then be analyzed as a group and served in an order determined by a scheduling policy. To support this design, the thesis also explores whether SCTP provides a more natural abstraction than TCP for implementing structured

web downloads.

1.3 Objectives

The main objectives of the thesis are as follows:

1. To design and implement a framework that enables flow-level scheduling for HTTP requests, using prefetching to extract collections of resources from HTML documents.

2. To build the server-side infrastructure in Rust, using either TCP or SCTP as transport protocols, and implementing scheduling policies.

3. To evaluate the performance of this approach via benchmark tests, comparing SCTP-based servers with their TCP equivalents.

Chapter 2

2 Related Work

2.1 Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP), introduced in RFC 793 [15], is a reliable and connection-oriented transport layer protocol that ensures in-order byte stream delivery between endpoints identified by IP address and port. TCP establishes connections via a three-way handshake and guarantees data integrity using sequence numbers, acknowledgments, and retransmissions [18].

Its reliability is further supported by flow control and congestion control mechanisms, including sliding windows and algorithms such as slow start and congestion avoidance [1]. TCP headers include critical metadata such as ports, sequence numbers, flags, and checksums to manage communication.

Despite its maturity and widespread support, the TCP single-stream in-order delivery model introduces limitations for modern applications requiring concurrent or multiplexed data flows such as head of line (HOL) blocking [18].

2.2 Stream Control Transmission Protocol (SCTP)

The Stream Control Transmission Protocol (SCTP) was standardized by the IETF in RFC 4960 [19], published in 2007. Initially developed for transporting signaling messages in telecommunication systems, particularly in the context of the SS7 stack over IP networks,

SCTP has since been proposed as a general-purpose transport layer protocol that combines the reliability of TCP with new features designed to improve robustness and performance.

Unlike TCP's three-way handshake, SCTP uses a four-way handshake to establish associations. This handshake involves a cookie mechanism that provides protection against SYN flooding attacks, a known vulnerability in TCP. When an endpoint receives an INIT message, it replies with an INIT-ACK containing a cookie. The association is only established once the peer returns the cookie in a COOKIE-ECHO message, which is then acknowledged. This design ensures that server resources are not allocated until the initiating peer proves that it received the cookie, effectively mitigating spoofed connection attempts [18].

SCTP offers several advantages over TCP, most notably its support for multistreaming [12]. While TCP transmits all data over a single ordered byte stream, SCTP allows data to be partitioned across multiple independent streams within a single association. This architecture significantly reduces the impact of head-of-line (HOL) blocking, as message delays in one stream do not affect others. Furthermore, SCTP is message-oriented, preserving message boundaries, and eliminating the need for the application to implement its own framing logic, as is often required with TCP. In addition to multistreaming, SCTP also supports multihoming, allowing endpoints to be associated with multiple IP addresses. This provides resilience to network failures by enabling seamless failover between network paths. Like TCP, SCTP is reliable and congestion aware: it ensures in-order delivery within each stream, uses acknowledgments and retransmissions, and implements congestion control and flow control mechanisms based on selective acknowledgments and window-based flow regulation.

Despite these technical advantages, SCTP remains underused in practice due to limited support in operating systems, middleware, and network infrastructure such as NATs and firewalls. Nonetheless, its unique features make it a promising alternative to TCP for applications requiring parallel, reliable, and congestion-controlled data flows.

2.3 Hypertext Transfer Protocol 1.1 (HTTP/1.1)

HTTP/1.1, standardized in RFC 2616 [3], has been the backbone of web communication since 1999. It introduced persistent connections and pipelining to reduce overhead and improve performance, along with enhancements such as chunked transfer encoding and more advanced caching and content negotiation [4].

However, its reliance on TCP creates head-of-line (HOL) blocking in persistent connections, particularly with pipelining, where a delayed response can stall others. Being text-based and lacking multiplexing, HTTP/1.1 struggles with efficient parallel resource delivery.

Although newer protocols such as HTTP/2 and HTTP/3 address these issues, HTTP/1.1

remains prevalent, making it a relevant topic for exploring alternative transport protocols such as SCTP.

2.4 SCTP developer tools

SCTP support on Linux systems is provided primarily by the lksctp-tools project lksctp-tools [10]. This collection of tools and libraries includes command-line utilities such as sctp_test and sctp_darn, as well as the libsctp library, which exposes the SCTP API for user-space applications.

Another important tool in the SCTP ecosystem is usrsctp [21], a user-space implementation of the SCTP protocol stack [14]. Originally developed by Google, usrsctp allows SCTP to run over UDP sockets, enabling SCTP-based communication even in environments where native kernel-level SCTP support is limited or unavailable. This approach is particularly useful in scenarios that involve containerized deployments, as the implementation of the SCTP kernel is not natively supported by containerization tools such as Docker [2]. By encapsulating SCTP packets within UDP, usrsctp provides a portable solution that can traverse NATs and firewalls more easily than native SCTP.

2.5 Request Scheduling

An important direction for improving web server performance is the scheduling of incoming requests. Rather than processing them in arrival order, modern systems can prioritize requests based on estimated cost or size. This approach enables faster response times for a larger number of clients, especially under high load. A notable scheduling policy in this area is the concept of size-based scheduling, which prioritizes shorter jobs to reduce the mean response time [6].

The motivation behind such policies also aligns with Zipf's law of least effort [25], which explains that users tend to request a small subset of popular and often smaller resources more frequently. These natural access patterns support the idea that scheduling policies should take advantage of request size and frequency to optimize performance.

2.6 Enhancing web servers and file transfer

Numerous optimizations have been explored to improve Web server performance, including techniques such as Locality-Aware Request Distribution (LARD) in distributed servers [23], as well as adopting alternative transport protocols such as SCTP, which has shown the potential to increase throughput by enabling efficient thread scheduling for handling concurrent streams across multicore systems [22].

SCTP also brings about improvements in file handling. Its message-oriented communication model allows file transfers to be segmented into discrete chunks, which can be

processed in parallel more efficiently than with TCP's byte-stream approach. This leads to reduced overhead and improved file-serving performance [9].

Chapter 3

3 System Architecture and Design

3.1 Preliminary Setup

Before presenting the detailed architecture, we reiterate the core objective: to design a Web server capable of leveraging SCTP as its transport layer protocol, in a way that aligns naturally with the concept of connection scheduling.

Since web browsers inherently rely on TCP for transport, a direct client-server implementation using SCTP is not feasible. Therefore, a TCP-to-SCTP proxy is introduced to bridge this gap. This proxy serves two essential functions. First, it compensates for the browser's lack of native SCTP support. Second, it ensures that the proposed system remains independent of any particular browser implementation, allowing general applicability.

The proxy establishes a single SCTP association with the server, configured with a fixed number of incoming and outgoing streams. This number typically matches the minimum number of CPU cores available on the proxy host or the server host. This alignment aims to reduce potential bottlenecks and facilitate efficient mapping between streams and worker threads.

On its TCP-facing side, the proxy listens for incoming connections from browsers clients and forwards their HTTP requests over the established SCTP association to the server.

On the server side, the SCTP-based system handles all requests received through the single association, applying a selected scheduling policy that governs how requests are distributed and served over the available streams.

3.2 Server Scheduling

Leveraging SCTP's multistreaming capabilities, incoming HTTP requests can be concurrently handled using one of the following scheduling policies.

3.2.1 Shortest Connection First (SCF)

The SCF policy is implemented on the server side and prioritizes sending responses to incoming `GET` requests based on the size of the requested files, serving smaller files first. Currently, SCF is applied to static content, but it can be trivially extended to dynamic

requests whose results are cached, such as when using key-value systems like Redis [17]. This strategy aims to reduce average response time and improve perceived performance in the browser by enabling lightweight resources to be rendered earlier.

The SCF policy is implemented on the server side and prioritizes sending responses to incoming `GET` requests based on the size of the requested files, serving smaller files first. This strategy aims to reduce average response time and improve perceived performance in the browser by enabling lightweight resources to be rendered earlier.

To support this behavior, the server maintains a priority queue that orders requests by file size. The queue is protected by a mutex and a condition variable to ensure safe concurrent access. A thread pool is initialized on the server, with its size equal to the number of outgoing SCTP streams negotiated in the association.

Each worker thread is assigned to a specific stream and waits for a new request to become available in the queue. Once a request is dequeued, the worker processes it and transmits the corresponding file through its associated stream. See figure 3.1.

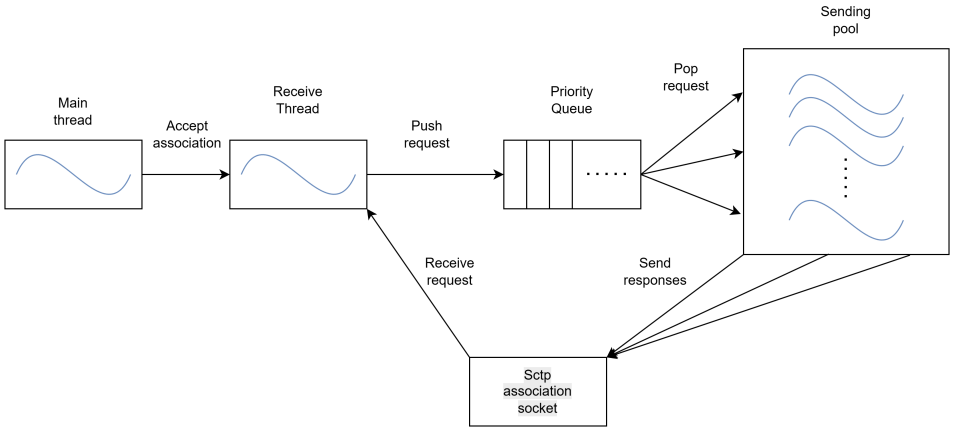


Figure 3.1: SCF architecture

3.2.2 Round Robin (RR)

The Round Robin policy distributes incoming requests across available SCTP streams in a cyclic manner, without any consideration for request characteristics such as file size. A thread pool is again employed by the server, with one worker per stream. Each request is assigned to a worker thread in round-robin fashion, ensuring a balanced distribution of workload across streams. See figure 3.2.

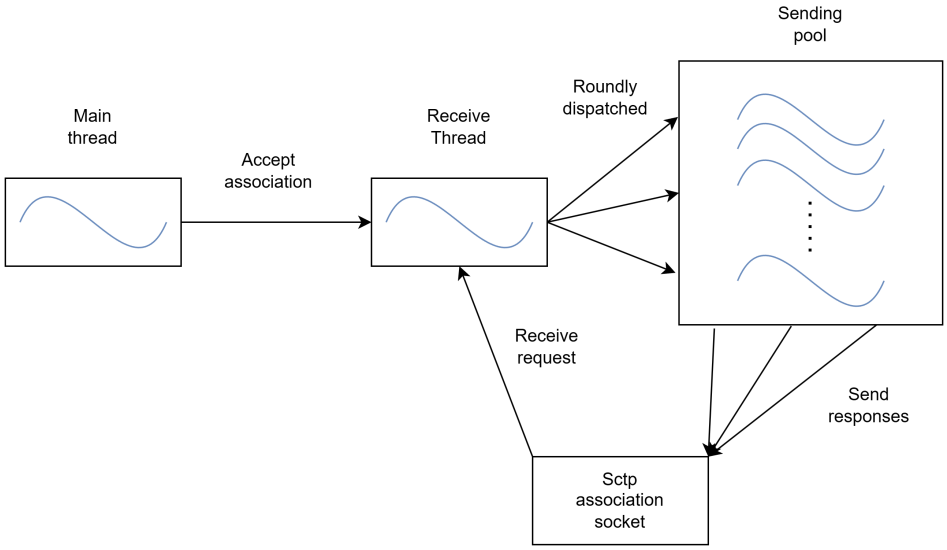


Figure 3.2: RR architecture

3.3 Proxy Design

The TCP-to-SCTP proxy acts as a separate endpoint for communication between the browser client and the SCTP-based server. Its primary role is to bridge the transport layer incompatibility, enabling standard TCP-based web clients to interact with the SCTP infrastructure. See figure 3.3.

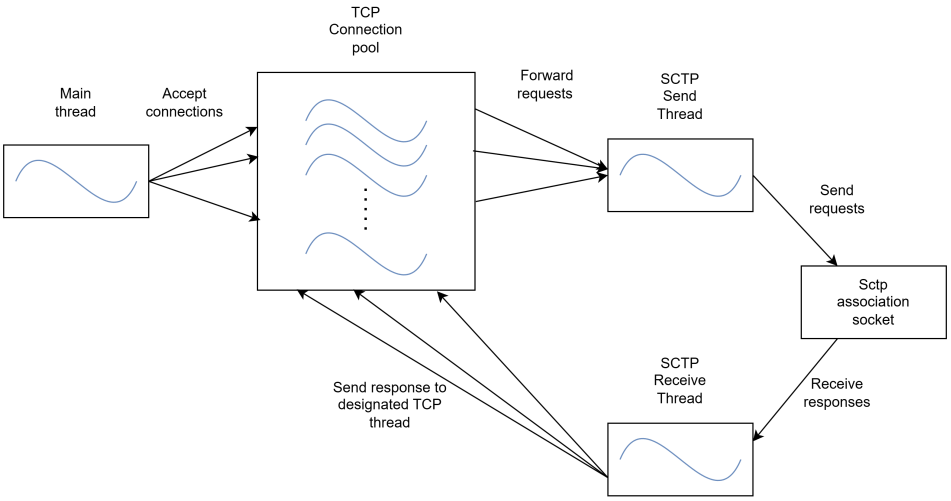


Figure 3.3: Unoptimized proxy architecture

Although introducing an additional endpoint may introduce some latency overhead, it also creates opportunities for optimization. Since the proxy operates on the application layer and has access to the full content of HTTP requests and responses, it can implement caching mechanisms and exploit the predictable structure of HTML documents.

By parsing HTML responses, the proxy can identify static assets(images, stylesheets or scripts) and prefetch them, anticipating future client requests and improving perceived performance.

Thus, the enhanced proxy design consists of two separate processes that run on the same machine. One process will handle the SCTP related computing of the proxy and the other will handle the TCP related computing. While the browser does not need to run on the same machine as the proxy, in typical deployments they are colocated. However, since the proxy can independently respond to TCP requests, it can also be deployed remotely, in which case it begins to function similarly to a front-end server. See figure 3.4.

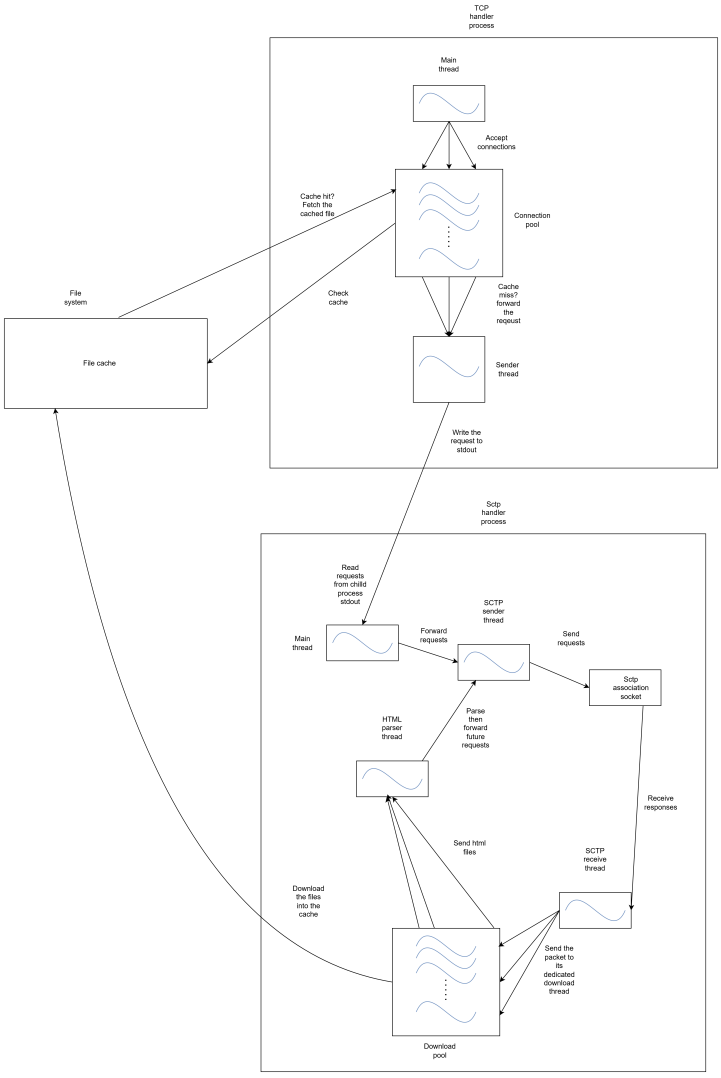


Figure 3.4: Proxy Design with caching and prefetching

- The TCP process opens the TCP server that will handle HTTP clients. On every browser request, a global in-memory cache is checked. Cache hits result in immediate transfer of the data to the client browser, while cache misses result in sending the request to the parent SCTP process via standard output.

- The SCTP process opens a persistent association to the server and creates the TCP child process. Each file fetched by this process is downloaded in the in-memory cache using a download thread-pool. Each time an HTML file is downloaded, a separate thread parses it and sends the requests of each embedded file path to the server so that it can be fetched before the client requests it.

The effect of the prefetching strategy is that it groups together multiple client-side requests into a coherent batch, which is transmitted to the server as a single logical unit. This enables the server to perform scheduling decisions not just per request, but over an entire collection of requests, allowing policies such as Shortest Connection First to prioritize smaller files and reduce perceived latency. By combining application-level insight with transport layer stream control, the system creates a feedback loop between the proxy and the server that promotes efficient scheduling across all streams.

This architectural separation allows the proxy to act not only as a transport bridge but also as a smart middleware component that manages traffic in a way that is optimized for the server's scheduling capabilities.

Chapter 4

4 Implementation

4.1 Overview

For implementing the systems discussed in the last chapter, a UNIX-based operating system is required that enables the SCTP protocol module. To enable SCTP support, the package lksctp-tools was required to be installed, which offers a user-friendly SCTP API. The codebase is mainly written in Rust, a language that provides strong guarantees of memory safety and leverages zero-cost abstractions, ensuring that high-level constructs introduce no additional runtime overhead compared to equivalent low-level code [8]. The implementation is heavily dependent on Rust's standard library, as it uses the data structures, net, and multi-threading tools.

4.2 SCTP and Low-Level Linux APIs in Rust

SCTP support in Rust was achieved by creating C bindings using a foreign function interface (FFI), which allows Rust code to interoperate directly with the system's native SCTP API through the C Application Binary Interface (ABI). Once all the needed SCTP C dependencies are ready to be used, soft wrappers were created over the API so that it can be used in a safe, aligned with Rust compile-time safety guarantees.

Additional low-level system calls, including *mmap*, *epoll*, *inotify*, *listen* and *dup2*, were accessed through well-maintained Rust libraries that internally rely on FFI (memmap2, mio, inotify, libc). These libraries provide safe, idiomatic Rust abstractions over the underlying Linux APIs, allowing seamless integration while preserving memory safety and concurrency guarantees enforced by the Rust compiler.

4.3 SCTP Server and Request Scheduling

The SCTP server reads its runtime parameters from a configuration file, allowing customization of options such as the listening port, IP address, buffer sizes, the maximum number of incoming and outgoing SCTP streams, the scheduling policy to be used, and the root directory from which the server serves files in response to HTTP requests. Based on these parameters, an SCTP listening socket is initialized. The server then enters a loop in which it accepts incoming associations. Each accepted association is handled on the main thread according to the scheduling policy selected at startup.

4.3.1 Shortest Connection First Scheduling

Once an SCTP association is accepted, its associated stream is passed to the scheduler service. The scheduler spawns one dedicated receiver thread and a fixed number of sender threads, equal to the number of outgoing SCTP streams available in the association. A shared, lock-protected min-heap is initialized as the request queue, implemented using a reference-counted smart pointer, and guarded by a mutex and a condition variable.

The receiver thread continuously processes incoming requests from the client, maps the requested files into memory using the *mmap* system call, and inserts the resulting memory-mapped structures into the heap. The heap automatically sorts requests in ascending order of file size, allowing smaller files to be prioritized.

Each sender thread attempts to acquire access to the heap and waits on the condition variable if the queue is empty. Once a request becomes available, the sender retrieves it, then transmits the file in chunks over a designated, preassigned SCTP stream. See Figure 3.1.

4.3.2 Round Robin Scheduling

In this variant, the scheduling logic follows a fair, stream-based distribution model. As with SCF, the scheduler service spawns one receiver thread and a number of sender threads equal to the outgoing stream count.

Here, the receiver thread maintains an internal round-robin counter to distribute requests across the available sender threads. To enable communication, each sender thread is associated with a separate unbounded channel, and the receiver thread holds the corresponding transmitter ends.

As HTTP requests are received, they are mapped into memory using the *mmap* system call and then dispatched in round-robin fashion to the sender threads via the respective channels. Each sender thread transmits the file it receives using its assigned SCTP stream. This approach ensures a uniform load distribution. See Figure 3.2.

4.4 SCTP Proxy, Caching and Prefetching

The SCTP proxy is implemented using a dual-process architecture, where one process manages TCP connections and the other maintains the SCTP association with the server. Communication between the two processes is achieved via a unidirectional *pipe*: the TCP process forwards HTTP requests to the SCTP process through its standard output, while the SCTP process notifies the TCP process of completed responses using the *inotify* kernel API. The downloaded responses are stored in a memory-backed temporary cache mounted via *tmpfs*, enabling fast access and reducing disk I/O [20]. See figure 3.4.

4.4.1 SCTP Process

Before the SCTP process starts, a shell script ensures that the cache directory is mounted as a tmpfs filesystem, allowing in-memory file storage. Upon startup, the SCTP association is established and the system initializes the following components.

- An SCTP sender thread, which receives requests from an unbounded channel and forwards them to the server using a round-robin strategy across the association's outgoing streams.
- An SCTP receiver thread, responsible for collecting incoming chunks from the server and dispatching them to the appropriate worker thread based on the stream number.
- A pool of download worker threads, with one thread per incoming SCTP stream. Each thread receives chunks from the SCTP receiver and writes them to files within the cache. To ensure safe concurrent access, each worker writes to only one file at a time. Downloaded files are initially created with a `.tmp` suffix to indicate that they are in progress. Once an HTML file is fully downloaded, a signal is sent to the HTML parser thread.
- An HTML parser thread, which processes completed HTML files to extract embedded dependencies (such as images, stylesheets, or scripts). These dependencies are then sent in advance to the server by forwarding them to the SCTP sender thread for prefetching.
- The TCP process is launched as a child process using Rust's `Command` API, which offers a safe and controlled way to generate processes from external executables.

The main thread of the SCTP process then takes responsibility for reading incoming requests from the pipe connected to the child's standard output.

4.4.2 TCP Process

As mentioned above, the TCP process is launched by the SCTP parent process. At runtime, the following architecture is initialized:

- A thread pool for handling browser connections, with one thread per available CPU core. A shared unbounded channel is used to distribute incoming connections to the threads in the pool.
- A dedicated HTTP request writer thread, responsible for serializing and forwarding all cache-miss requests to standard output. This approach avoids contention by eliminating the need for each connection handler to lock access to standard output.
- A global, shared hash map that tracks all files currently being downloaded. Each entry maps a file path to a mutex and a condition variable, enabling threads to wait until the file is ready.
- An *inotify* thread that monitors the cache directory for `MOVE` events. As mentioned above, downloading files are initially named with a `.tmp` suffix. Once a file is completed, it is renamed, triggering a `MOVE` event. The *inotify* thread identifies the affected file, retrieves the corresponding condition variable from the global map, and notifies all threads waiting for that file.

Once the setup is complete, the TCP process's main thread begins listening for client connections. For each accepted connection, a thread from the pool handles the HTTP request in one of two scenarios:

- **Cache hit:** If the requested file is already present in the cache, it is immediately mapped to memory using the *mmap* system call and sent to the client.
- **Cache miss:** If the file is not present, the request is forwarded to the HTTP writer thread via an unbounded channel. The connection thread then registers a new entry in the global hashmap, associating the file path with a mutex and condition variable. Finally, the thread blocks, waiting on the condition variable until the file is fully downloaded and available.

Chapter 5

5 Benchmarking Results

5.1 Setup

As the SCTP protocol is not supported in containerization tools such as Docker and virtual machines introduce unnecessary overhead, it is required that the benchmarking ecosystem be distributed on 3 physical devices on a local network. Therefore, a Raspberry Pi 5 Model B, having a 2.4 GHz 4-core CPU and 8GB of LPDDR4 RAM will host the SCTP server. The SCTP Proxy will run on a Lenovo IdeaPad laptop, featuring an AMD Ryzen 5 5600H 12-core CPU and 32 GB of DDR4 RAM. The TCP client that shall connect to the proxy will run separetely on a Synology DS923+ NAS with a 2.6 GHZ AMD Ryzen R1600 4-core CPU and 16GB of DDR4 RAM.

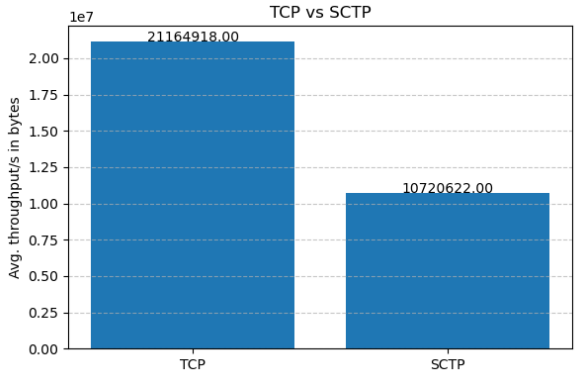
The benchmark scripts were written in Python using a powerful load testing tool called Locust [11]. The load generation was achieved by creating custom user scripts that simulate HTTP clients sending requests to the server. Performance measurements were dynamically stored in CSV (Comma-Separated Values) files, which could later be processed using data manipulation tools such as Pandas [13] and visualized using libraries such as Matplotlib [7]. These tools allowed for efficient analysis and presentation of performance metrics such as throughput and latency under various test scenarios.

5.2 Performance Tests

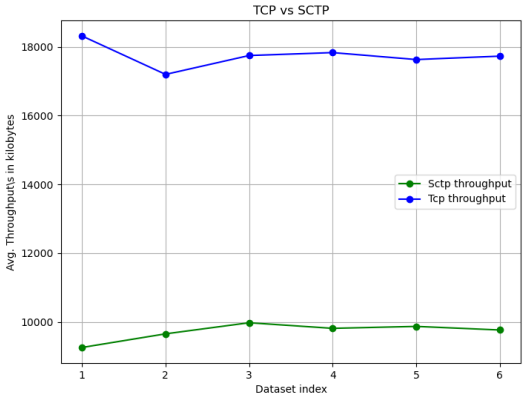
5.2.1 Single-stream TCP vs single-stream SCTP

A first set of benchmarking experiments compares the performance of TCP and SCTP when used by two HTTP servers that share the same architecture but differ in the transport layer protocol. Each server handles a single connection: one using a TCP stream and the other using a single SCTP association configured to use just a single stream for message exchange. The goal is to measure the performance of the two protocols. The unit of measure will be the average throughput of each server in a fixed amount of time. To make sure the tests are as fair as possible, each server follows the same fetching method, using the *mmap* system call and chunked file sending of fixed size using the send call on each socket. Custom clients were also written that follow the same payload reconstruction method: making use of the ”Content-Length” header of the http protocol.

In an environment where the servers have to process the same 10.000 requests of random sized files, the results are shown in figure 5.1.



(a) 1-stream TCP vs 1-stream SCTP over 10.000 requests



(b) 1-stream TCP vs 1-stream SCTP over a set of 60.000 requests

Figure 5.1: Throughput comparison between TCP and SCTP for single-stream communication.

Surprisingly, TCP outperformed SCTP in terms of raw throughput, having a nearly 100% increase in performance. This result was unexpected and notable, considering the theoretical advantages of SCTP.

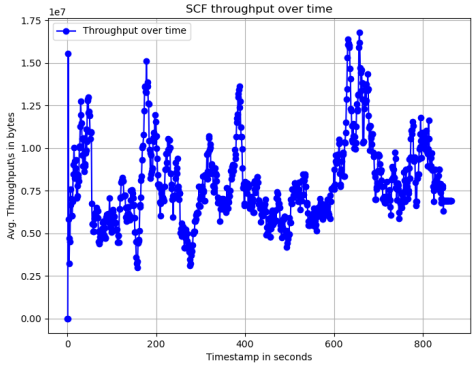
After several attempts to tune SCTP socket options and kernel buffer sizes, no significant improvements were obtained. Therefore, we accept as a baseline premise that TCP performs better than SCTP in this configuration. We attribute this to TCP's maturity and decades of optimization.

5.3 Scheduling Policies in SCTP-based Servers

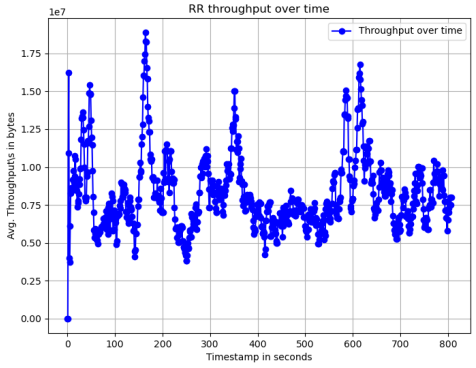
After establishing the baseline performance comparison between TCP and SCTP for single-stream communication, the focus of this section is on evaluating different scheduling strategies in an SCTP-only server environment. The goal is to explore whether request scheduling policies can affect overall throughput or improve responsiveness in specific scenarios.

To ensure fairness and avoid introducing additional optimizations that could alter the results, the unoptimized version of the TCP-to-SCTP proxy was used which was discussed in earlier chapters 3.3. The benchmark workload consisted of 6.000 identical file requests distributed evenly on the TCP connections.

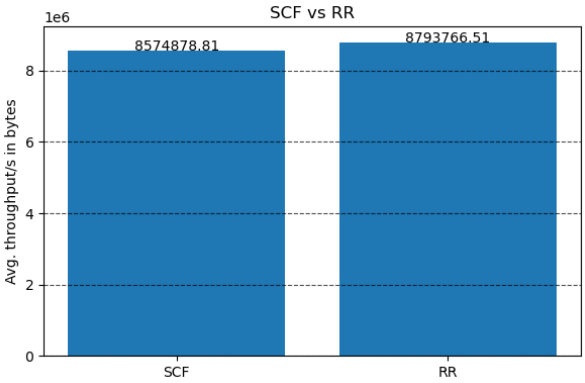
Two scheduling strategies compared are SCF(Shortest Connection First) 3.1 and RR(Round Robin) 3.2.



(a) Shortest Connection First throughput over time



(b) Round Robin throughput over time



(c) Average scheduling throughput

Figure 5.2: Throughput comparison between SCF and RR scheduling.

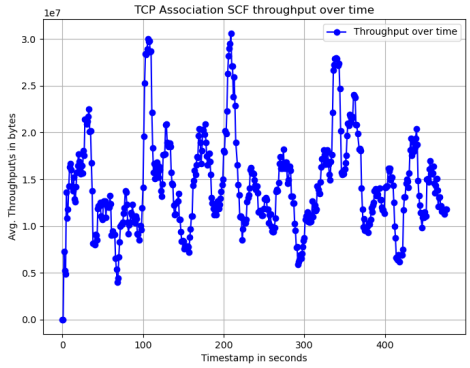
The benchmark results show that both scheduling strategies yield similar throughput under the tested conditions, where requests were distributed randomly rather than following a Zipf-like distribution typical of real-world web traffic. This result is encouraging, as it suggests that Shortest Connection First (SCF) performs on par with Round Robin (RR) even in non-ideal conditions; according to prior research on size-based scheduling, SCF tends to outperform RR when request sizes follow Zipfian patterns. Thus, SCF remains a safe and potentially beneficial choice for realistic workloads, as also supported in the literature [6].

5.4 Simulating SCTP over TCP

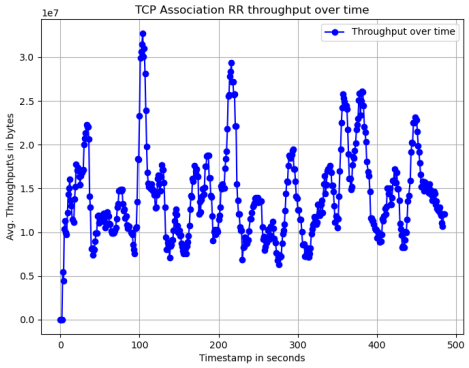
Given the lack of significant performance differences between the tested scheduling policies, further testing was needed to check wheather the inherent limitations of the SCTP kernel implementation might be the primary bottleneck. Therefore, an experimental wrapper was developed to emulate SCTP-like behavior over multiple TCP connections. This approach involved creating a lightweight, message-based transmission layer in user

space, using standard TCP sockets as the underlying transport.

The custom protocol mimics key features of SCTP associations: it allows messages to be explicitly sent over specific streams, and when reading from the connection, it retrieves messages from any ready stream, enabling independent stream delivery similar to SCTP's multistreaming capabilities. The architecture of the system remains the same, with slight changes: the server will communicate with the proxy using a TCP-association.



(a) Shortest Connection First throughput over time



(b) Round Robin throughput over time

Figure 5.3: SCF and RR scheduling throughput over time.

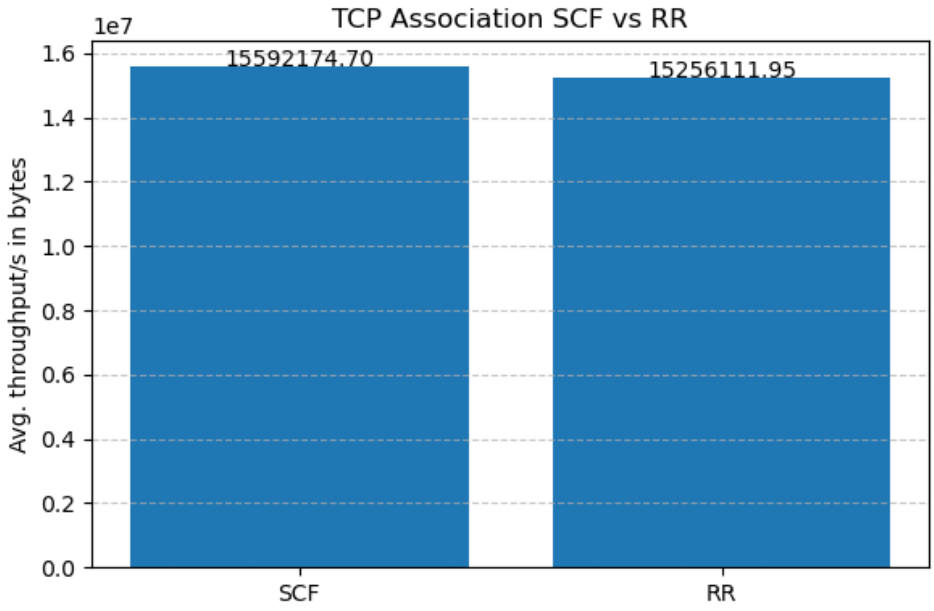


Figure 5.4: Average TCP association scheduling throughput comparison.

Benchmarking was repeated in this configuration, using the same request patterns and workloads as in previous tests. The results showed that, once again, there were no significant performance differences between the SCF and Round Robin scheduling policies

under the emulated SCTP-over-TCP setup. However, the overall throughput of this configuration remained roughly twice as high as that of the native SCTP implementation, mirroring the results observed in the initial TCP vs. SCTP benchmarks.

These findings reinforce the hypothesis that the current SCTP kernel implementation is a limiting factor in performance. At the same time, they confirm that the choice of scheduling strategy, whether SCF or RR, does not significantly affect throughput under a random and uniform request distribution. This suggests that SCF does not introduce performance penalties in non-Zipf scenarios.

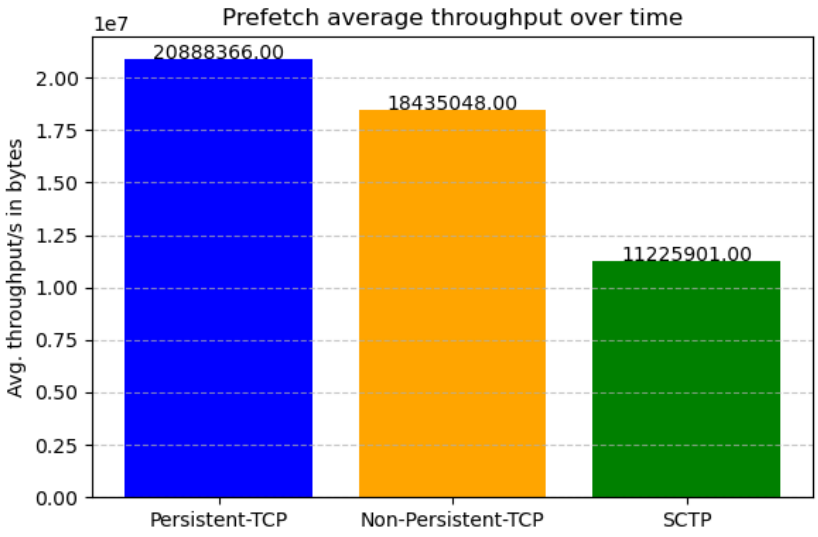
5.5 Simulating browser request patterns and prefetching behaviour

In this experiment, we simulate a realistic browsing scenario by modeling how modern browsers, such as Google Chrome, initiate requests for embedded resources. Specifically, browsers tend to request resources in the order in which they appear in the parent HTML document. To replicate this behavior, our testing framework preprocesses HTML documents and immediately schedules requests for all embedded resources after each HTML file is fetched.

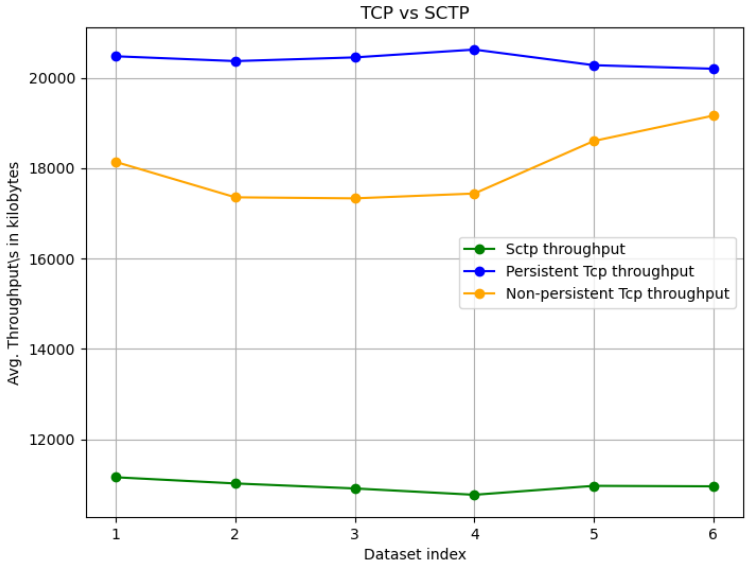
To fully evaluate the system under different network configurations, the benchmark covers three server variants:

- A classic TCP server using non-persistent connections, where each request opens a new connection.
- A TCP server using persistent connections, reusing the same socket for multiple requests.
- The SCTP server, which uses concurrent streams within a single association.

In all three setups, requests are distributed across multiple concurrent threads, each using its own stream (in SCTP) or connection (in TCP), thus ensuring parallelism similar to what browsers typically exploit. The proxy prepares a queue of 5000 requests per test, with each HTML file followed by its embedded resources. The same workload is applied across all test classes to ensure comparability. Each configuration is tested across 6 separate runs of 5000 requests to ensure statistical stability.



(a) Average throughput in bytes/s for all 30.000 requests



(b) Throughput in kilobytes/s across 6 runs of 5.000 requests

Figure 5.5: Throughput comparison for different prefetching scenarios

The results show that SCTP maintains consistent throughput due to parallel delivery via streams, but overall remains approximately twice as slow as the TCP server, both in persistent and non-persistent modes as previously shown in 5.2.1.

Chapter 6

6 Future Work

6.1 Threading model improvement

The current implementation relies on a 1:1 threading model, where each connection or task is handled by a dedicated system thread. Although this approach is functional and straightforward, it introduces potential scalability limitations, especially under high concurrency workloads. A promising future improvement would be to replace the current model with an asynchronous runtime that leverages green threads(futures or coroutines). This change would allow the system to manage a large amount of concurrent tasks within a single thread, enabling more efficient use of system resources and reducing context-switching overhead.

6.2 Solving SCTP incompatibilities

A further improvement would be to replace the kernel-level SCTP API with Google's user-space implementation usrsctp. Since native SCTP support is often limited in containerized environments and struggles with NAT traversal issues, using a user-space implementation of SCTP over UDP would provide greater flexibility and compatibility. This change would enable the deployment of the server and proxy components within Docker containers and Kubernetes clusters, facilitating scalability and easier orchestration in modern distributed environments.

6.3 Horizontal scaling

Having improved the threading model, allowing vertical scaling of the handling tasks and clients, an important step forward would be to explore the possibility of horizontal scaling by introducing multiple network nodes running the server and the proxy.

A potential improvement would involve deploying multiple server instances and load-balancing traffic across them. Similarly, introducing additional proxy nodes could improve fault tolerance and throughput by distributing client requests more efficiently. This multi-node architecture would enable the system to handle higher traffic loads and improve resilience to node failures, paving the way for more complex scenarios and production-like deployments.

Chapter 7

7 Conclusions

The main objective of this thesis was to design, implement and evaluate the performance of a system capable of stream-level scheduling in a web server environment. The proposed architecture is built on top of the Stream Control Transmission Protocol (SCTP), which provides a flexible API for managing multiple independent data streams within a single connection. This native support for multistreaming enabled the implementation of scheduling policies at the transport layer, without requiring changes to the HTTP/1.1 application protocol.

In order to enrich the system's functionality, a TCP-to-SCTP proxy was developed. The proxy handles all client communication and supports performance-enhancing mechanisms such as prefetching and caching. This design allows legacy HTTP clients to interact with the SCTP server transparently, while enabling more intelligent traffic management on the server side.

The evaluation began by establishing a baseline comparison between TCP and SCTP, focusing on raw performance in one-to-one communication scenarios. This initial benchmark provided a reference point for interpreting the impact of additional system features. Building on this baseline, further experiments were conducted to isolate and assess the contribution of individual components, such as stream scheduling on the server and prefetching mechanisms in the proxy. All evaluations were conducted in parallel with equivalent TCP-based setups to ensure a consistent frame of reference.

Although the results confirmed that TCP remains significantly faster in most scenarios, particularly due to the mature and optimized kernel implementation, the SCTP-based approach demonstrated predictable and stable behavior under scheduling policies.

Several limitations remain, most notably the poor performance of the current SCTP implementation in the Linux kernel. However, potential fixes have been discussed in chapter 6. These improvements could help unlock the full potential of SCTP in web server architectures and pave the way for further experimentation in the area of transport-layer scheduling.

References

[1] Mark Allman, Vern Paxson, and Ethan Blanton. *TCP Congestion Control*. RFC 5681. Sept. 2009. URL: https://www.rfc-editor.org/rfc/rfc5681.html.

[2] *Docker: Empowering App Development for Developers*. URL: https://www.docker.com/.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*. https://www.rfc-editor.org/rfc/rfc2616. Internet Engineering Task Force. 1999.

[4] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. https://www.rfc-editor.org/rfc/rfc7230.html. Internet Engineering Task Force. June 2014.

[5] Răzvan Petru Gabroveanu. *Communication Flow Scheduling in a Web Server - Source Code*. URL: https://github.com/GabroveanuRazvan/Web-Server-Communication-Flow-Scheduling.

[6] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. "Size-Based Scheduling to Improve Web Performance." In: *ACM Transactions on Computer Systems* 21.2 (May 2003), pp. 207–233. URL: https://cs.toronto.edu/~bianca/papers/submtocs-3.pdf.

[7] John D. Hunter and The Matplotlib Development Team. *Matplotlib: Visualization with Python*. URL: https://matplotlib.org/.

[8] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. San Francisco, CA, USA: No Starch Press, 2019. URL: https://doc.rust-lang.org/book/.

[9] S. Ladha and P.D. Amer. "Improving file transfers using SCTP multistreaming." In: IEEE. 2005. URL: https://ieeexplore.ieee.org/abstract/document/1395080.

[10] *lksctp-tools: Linux Kernel SCTP Project*. URL: https://github.com/sctp/lksctp-tools.

[11] *Locust: Scalable User Load Testing Tool*. URL: https://locust.io/.

[12] Preethi Natarajan, Janardhan R. Iyengar, Paul D. Amer, and Randall Stewart. "SCTP: An innovative transport layer protocol for the web." In: *Proceedings of the 15th International Conference on World Wide Web*. Edinburgh, Scotland: ACM, 2006, pp. 615–624. URL: https://dl.acm.org/doi/10.1145/1135777.1135867.

[13] *Pandas: Data Analysis Library*. URL: https://pandas.pydata.org/.

[14] Brad Penoff, Alan Wagner, Michael Tüxen, and Irene Rüngeler. "Portable and Performant Userspace SCTP Stack." In: *2012 21st International Conference on Computer Communications and Networks (ICCCN)*. 2012. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6289222.

[15] J. Postel. *RFC 793: Transmission Control Protocol*. https://www.rfc-editor.org/rfc/rfc793. Internet Engineering Task Force. 1981.

[16] Linux man-pages project. *Linux Programmer's Manual - Section 2: System Calls*. https://man7.org/linux/man-pages/man2/.

[17] Redis Ltd. *Redis*. URL: https://redis.io.

[18] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming: The Sockets Networking API*. 3rd. Addison-Wesley Professional, 2004.

[19] R. Stewart. *RFC 4960: Stream Control Transmission Protocol*. https://www.rfc-editor.org/rfc/rfc4960. Internet Engineering Task Force. 2007.

[20] The Linux Kernel Team. *tmpfs - a virtual memory filesystem*. https://www.kernel.org/doc/Documentation/filesystems/tmpfs.rst.

[21] *usrsctp: A Portable User-Space SCTP Stack*. URL: https://github.com/sctplab/usrsctp.

[22] Olaru Vlad, Mugurel Andreica, and Nicolae Tapus. "Using the Stream Control Transmission Protocol and Multi-core Processors to Improve the Performance of Web Servers." In: IEEE. 2011. URL: https://ieeexplore.ieee.org/document/6062986.

[23] Olaru Vlad and W.F. Tichy. "On the design and performance of kernel-level TCP connection endpoint migration in cluster-based servers." In: IEEE. 2005. URL: https://ieeexplore.ieee.org/document/1558670.

[24] Olaru Vlad and W.F. Tichy. "Request distribution-aware caching in cluster-based Web servers." In: IEEE. 2004. URL: https://ieeexplore.ieee.org/abstract/document/1347792.

[25] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort*. Cambridge, MA: Addison–Wesley Press, 1949. URL: https://archive.org/details/in.ernet.dli.2015.90211.