UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

TRANSPARENT MECHANISM FOR THE COMPRESSION OF NETWORK FLOWS

Diploma Thesis

Vasileios Kyriakakis

Supervisor: Christos Antonopoulos

June 2023

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

TRANSPARENT MECHANISM FOR THE COMPRESSION OF NETWORK FLOWS

Diploma Thesis

Vasileios Kyriakakis

Supervisor: Christos Antonopoulos

June 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΑΦΑΝΗΣ ΜΗΧΑΝΙΣΜΟΣ ΣΥΜΠΙΕΣΗΣ ΔΙΚΤΥΑΚΩΝ ΡΟΩΝ

Διπλωματική Εργασία

Κυριακάκης Βασίλειος

Επιβλέπων: Αντωνόπουλος Χρήστος

Ιούνιος 2023

Approved by the Examination Committee:

Supervisor	**Christos Antonopoulos** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Spyros Lalis** Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Nikolaos Bellas** Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Christos Antonopoulos for his guidance throughout the entire research process. I would also like to thank Professor Spyros Lalis for offering me advice and being a part of examination committee, and Professor Nikolaos Bellas for his contributions as a committee member.

To my parents Ioanna and Nikos and my siblings Stavroula, Sotiris and Anastasia, thank you for your unconditional love and support.

To my girlfriend Elena, thank you for being there for me in every step of this difficult process, while supporting me in every conceivable manner, no matter how difficult things were.

To my friends, Aris, Thodoris, Themis, and Vangelis thank you for the fun times and support.

To every other person that supported me during my studies, even if it was in a small way, even if we aren't in contact anymore, thank you.

Diploma Thesis

TRANSPARENT MECHANISM FOR THE COMPRESSION OF NETWORK FLOWS

Vasileios Kyriakakis

Abstract

Cloud computing as we know it is unable to cope with the enormous amounts of data produced in the network edge, especially considering the proliferation of IoT devices. The edge computing paradigm seeks to solve this issue by moving the data processing to the edge of the network, away from the cloud and closer to the entities that generate or consume the data. Because some of the links in the network edge can be slow, the large amounts of data exchanged might lead to decreased performance. In this Thesis, we designed and implemented a library that uses lossless data compression over TCP, in order to reduce the amount of transmitted data in the network edge and mitigate the effects of the low bandwidth edge links. We validated the correctness of our implementation and evaluated its performance using a platform we built over Mininet. From the experimental data, we conclude that our library performs better than plain TCP in low bandwidth and/or high packet loss rate networks and that when it is used in such networks, its CPU overhead is negligible.

Keywords:

cloud computing, edge computing, data compression, zlib, bandwidth, Mininet

Διπλωματική Εργασία

ΔΙΑΦΑΝΗΣ ΜΗΧΑΝΙΣΜΟΣ ΣΥΜΠΙΕΣΗΣ ΔΙΚΤΥΑΚΩΝ ΡΟΩΝ

Κυριακάκης Βασίλειος

Περίληψη

Το cloud computing, όπως το γνωρίζουμε, δεν μπορεί να ανταπεξέλθει στοΝ τεράστιο όγκο δεδομένων που παράγονται στο edge, ιδίως λόγω της ανάπτυξης των συσκευών IoT. Το edge computing επιδιώκει να λύσει αυτό το ζήτημα μεταφέροντας την επεξεργασία δεδομένων στο edge, μακριά από το cloud και πιο κοντά στις οντότητες που παράγουν ή/και καταναλώνουν τα δεδομένα. Επειδή κάποιες από τις ζεύξεις στο edge είναι αργές, οι μεγάλες ποσότητες δεδομένων που ανταλλάσσονται μπορεί να οδηγήσουν σε μειωμένη απόδοση. Σε αυτήν τη διπλωματική εργασία, σχεδιάσαμε και υλοποιήσαμε μια βιβλιοθήκη που χρησιμοποιεί συμπίεση δεδομένων χωρίς απώλειες πάνω στο TCP, στοχεύοντας να μειώσουμε την ποσότητα των δεδομένων που μεταδίδονται και να αντιμετωπίσουμε τις επιπτώσεις των ζεύξεων χαμηλής χωρητικότητας στο edge. Επιβεβαιώσαμε την ορθότητα της υλοποίησης μας και αξιολογήσαμε την απόδοσή της χρησιμοποιώντας μια πλατφόρμα που αναπτύξαμε πάνω στο Mininet. Με βάση τις μετρήσεις που προέκυψαν, καταλήξαμε ότι η βιβλιοθήκη μας παρουσιάζει καλύτερη απόδοση από το απλό TCP σε δίκτυα χαμηλής χωρητικότητας ή/και υψηλή πιθανότητα απώλειας πακέτων και ότι επιβαρύνει τη CPU σε αμελητέο βαθμό όταν χρησιμοποιείται σε τέτοιου είδους δίκτυα.

Λέξεις-κλειδιά:

cloud computing, edge computing, συμπίεση δεδομένων, zlib, χωρητικότητα δικτύου, Mininet

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

The migration of company resources into the cloud was accelerated due to the COVID-19 pandemic [1], and is still ongoing, with Gartner [2] forecasting that worldwide end-user spending on public cloud services will grow 20.7% to total $591.8 billion in 2023. Enterprises that move their operations into the cloud enjoy many benefits such as elasticity, cost-effectiveness, and the offload of infrastructure administration to the cloud service providers [3].

However, the centralized nature of cloud computing paired with the fact that network bandwidth has remained stagnant [4], causes the performance to degrade when very large amounts of data are sent to the cloud. Also, the proliferation of IoT devices [5], will lead to enormous amounts of data being generated at the edge of the network, rendering cloud computing as we know it unable to service them. Another problem with the cloud computing model is that the applications are less responsive the farthest from the data center the users are, due to WAN latency [6]. This is an issue for time-sensitive applications such as autonomous vehicles [7], which collect large amounts of data with their sensors that need to be processed in order for the system to make predictions, where a delay in the response could cause serious accidents.

The edge computing [4] paradigm promises to solve the above issues and more, by moving the data processing to the edge of the network, away from the cloud, and closer to the end users. Since the total traffic will be distributed among the edge nodes, there exists no central point which might get congested. Also, since the edge nodes will be closer to the users, the latency will be significantly reduced and the interactive services' quality will be improved.

However, this solution is not without its problems. Some links in the edge of the network are relatively slow, which is acceptable if the edge is populated mainly by users sending data to the cloud, but if processing nodes, which will receive large amounts of data, are to be created at the edge this might pose a problem. Replacing all the edge links with high-speed links is impractical, so we turned our attention to finding a software-based solution.

We hypothesized that the low bandwidth could be mitigated by reducing the amount of transmitted data that is necessary for the applications to operate. One way to achieve this is by compressing the data before sending it and decompressing it before delivering it to the recipient (where the sender or recipient could be the end-user or the edge processing node). The problem with this approach is that additional CPU cycles must be spent in order to compress or decompress the data, which could outweigh any performance improvements by reducing the transmitted data.

With all that in mind, we set out to implement our own data compression solution.

1.2 Contribution

In this Thesis, we aimed to reduce the amount of application data by introducing a lossless compression layer between the application and TCP, hoping to improve overall application performance. A number of problems had to be solved first, including finding a way for applications to use the proposed functionality without needing to be rewritten, and carefully designing the way the data is sent/received in order to not cancel-out the benefits of compressing it.

After we finished with the design and implementation of our solution, we validated its correctness and evaluated its performance along different metrics, as well as its resource utilization. This data was necessary in order to study the aforementioned CPU-bandwidth trade-off in various network conditions, corresponding to both fast and slow links.

We also compared its performance with that of plain TCP in different network conditions, which we emulated using an evaluation platform we built on top of Mininet.

Our contribution can be summarized as follows:

- We developed our compression-over-TCP library.

- We used the dynamic library call interception mechanism, in order for applications to be able to use our implementation through the Unix socket API.
- We validated the correctness of our implementation.
- We evaluated the performance and resource utilization of the library.
- We developed an evaluation platform that can test our implementation in various network conditions using Mininet.
- We compared the performance of our library to that of plain TCP in various network conditions.

1.3 Structure

In Chapter 2 we introduce various concepts that will be necessary for the comprehension of the methodologies in our Thesis. We outline the general architecture of our compression library in Chapter 3. We discuss implementation details in Chapter 4. We present and discuss the experimental results in Chapter 5. We discuss work related to ours in Chapter 6. Finally, Chapter 7 concludes the Thesis.

Chapter 2

Background

2.1 Terminology

2.1.1 Data compression

Data compression [8] is the process of transforming data into a different representation that uses less bits. Data compression algorithms can be broadly separated in two categories, those that perform lossless compression and those that perform lossy compression.

Lossless data compression allows the original data to be perfectly reconstructed from the compressed data in a process that is called decompression. This is possible thanks to the redundancy present in the data. Two broad categories of lossless compression algorithms are entropy coding and dictionary-based algorithms.

Entropy coding algorithms first create a statistical model for the symbols appearing in the data, and then use that model to map the symbols to bit sequences so that symbols that appear often are replaced by shorter sequences than those that appear rarely. Huffman coding [9] and arithmetic coding [10] are commonly used algorithms of this type.

Dictionary-based algorithms build a list of commonly occurring patterns in the data, then use it to substitute these patterns with their index in the list. Most algorithms of this type are based on the Lempel-Ziv algorithms, LZ77 [11] and LZ78 [12].

In lossy compression the size of the data is reduced by discarding part of it. This leads to a higher amount of data reduction compared to lossless compression, however the process is irreversible. The choice of which parts to discard is made so that the resulting data approximates the original for the purposes of the end user. The most popular lossy compression algorithm is DCT [13] (discrete cosine transform).

2.1.2 Compression ratio

The compression ratio is the ratio between the uncompressed size and compressed size of the same data. The higher the compression ratio is, the more efficient the compression was at decreasing the size of that data. In general, the more redundant the data, the more it will be compressed and the ratio will be higher.

2.1.3 Library

A library is a collection of functions and variables that can be used by programs through a well-defined interface. Libraries can be broken down into two categories: statically linked and dynamically linked.

When using a statically linked library, its contents are embedded into the final executable during the build process. This process is performed by the linker, a program that combines a group of object files into an executable program or library.

On the other hand, when using a dynamically linked library its contents are loaded into memory and linked when the program is executed. This is done by a component of the operating system, the dynamic linker.

2.1.4 Dynamic library interception

When we use the term interception in the text, what we mean is that instead of the dynamic library that is supposed to provide some functionality, another one is loaded first and its functions are accessed by the program instead. This is done to alter the functionality of programs using the original library, without having to rewrite them.

2.1.5 Bandwidth

The bandwidth of a computer network is the maximum rate of data transfer through it. It is typically measured in Mbps (Megabits / sec).

2.1.6 Effective bandwidth

The rate of application data transfer, which is also measured in Mbps (Megabits / sec). It might differ from the actual network bandwidth depending on the manner in which the application data is encoded to messages and transmitted.

For example, if the application data is compressed, the messages that are actually transmitted contain a larger amount of data, so the effective bandwidth may be higher than the actual bandwidth. Also, an application might produce data at a slower rate than the available bandwidth, or transmit control data as part of its function, in which case the effective bandwidth will be lower than the actual bandwidth.

2.1.7 Latency

The latency in a computer network is the time it takes for a bit of data to travel from the sender to the receiver.

2.1.8 Acknowledgment

The acknowledgment (ACK) is a message that is sent to the sender of a packet, to signify that the packet has been received successfully.

2.1.9 Round-trip time

The round-trip time (RTT) is the time required for a sent packet to reach its destination, plus the time required for the corresponding acknowledgment to reach the sender.

2.1.10 Packet loss rate

Packet loss occurs when one or more packets of data traveling across a computer network fail to reach their destination. Packet loss can be caused by errors in data transmission, or network congestion. The packet loss rate is the percentage of packets lost with respect to packets sent.

2.2 zlib

zlib is a software library that performs lossless data compression. It is an important part of many applications, and has many desirable qualities, such as an easy-to-use interface, control over the size reduction/speed trade-off by setting a compression level parameter, and thread-safety.

It uses the DEFLATE [14] compression algorithm, which first uses the dictionary-based LZ77 algorithm to replace commonly occurring strings with a <length, backward distance> pointer to their previous appearance. Afterwards, the Huffman coding algorithm is applied, mapping symbols to bit sequence in the manner described in Section 2.1.1. Both byte literals and pointers are counted as symbols for this purpose. The decompression is performed using the INFLATE algorithm.

2.3 Mininet

Mininet [15] is a network emulator which creates a network of virtual hosts, switches, controllers, and links.

Each virtual host or switch is a process with its own network interfaces, routing table, and ARP table (using Linux's network namespaces feature). All of these processes run on the same Linux kernel. The virtual hosts can run any Linux program, while the switches support the OpenFlow protocol for software-defined networking. Also, the switches need a controller to function.

The network is defined by the user, through a set of commands in a CLI environment (ran using the `mn` command) or programmatically using the Python API [16]. The user can insert hosts and switches (with their controllers) into the network, and connect their (virtual) network interfaces using links. The behavior of the links can be modified through parameters that correspond to physical properties such as bandwidth, packet loss rate, and latency.

Mininet is often used for the development of a network by simulating it and running test applications on the hosts without needing to actually wire the physical network to debug it or measure its performance. It can also be used in application development to test the performance of the applications themselves in networks with different topologies and link parameters with similar benefits.

Chapter 3

Design

In this chapter, we will describe the general architecture of our compression library, namely the modules of which it consists, as well as the way it interfaces with the user applications and TCP. Also, we will discuss the buffering logic used in the sending and receiving modules in an abstract manner (we will offer more detail in Chapter 4).

3.1 Placement in the network stack

As seen in Figure 3.1, the compression library lies between the application and TCP. Our library intercepts the calls made by the application to the libc socket functions and calls the original functions in the context of the compression/decompression process.

Specifically, when a `send()` call is intercepted, the data is compressed and then the original `send()` is called by the library to pass the data to TCP. Similarly, when a `recv()` call is intercepted, the original `recv()` is called to receive compressed data from TCP, which is then decompressed and delivered to the application.

The reason we decided on the above, instead of just implementing the library and providing an API to the applications, is that applications written using the libc socket API will be able to use the library's functionality without needing to be rewritten.

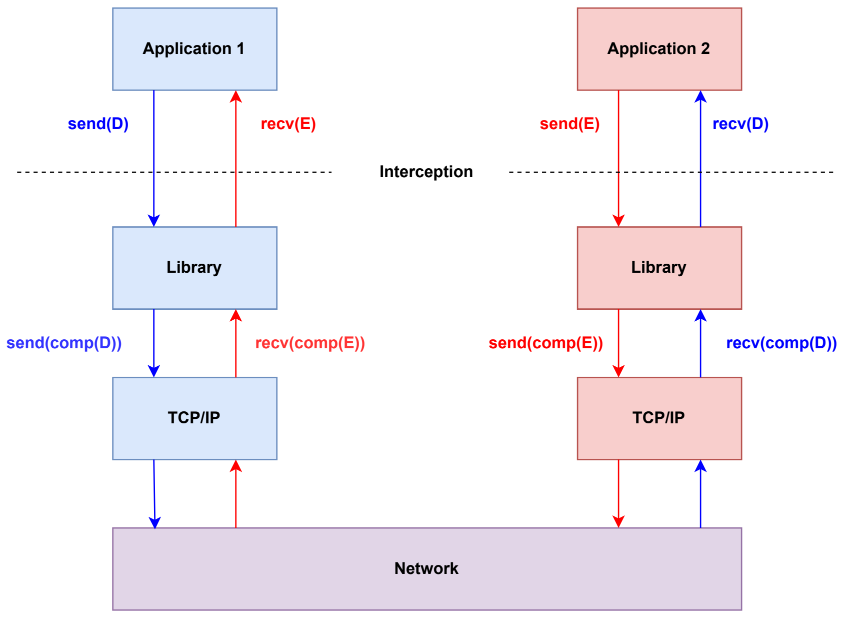

Figure 3.1: The library intercepts TCP socket calls to compress/decompress the data first.

3.2 Architecture

The library comprises two symmetric sides as shown in Figure 3.2. One side, the sending side, processes the intercepted `send()` calls, and the other one, the receiving side, processes the intercepted `recv()` call. We will explain the functionality of each side in the following subsections.

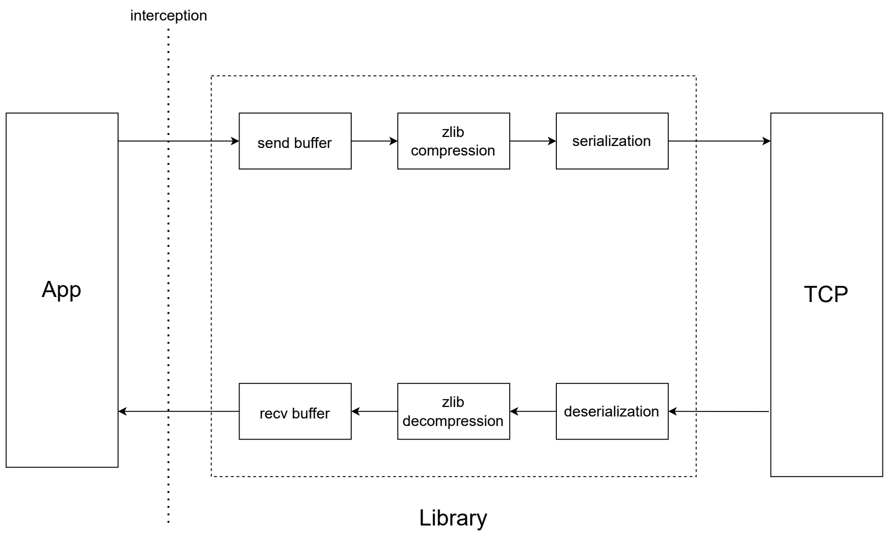

Figure 3.2: The sending and receiving sides of the library.

3.2.1 Sending side

The sending side receives data from the application and places it in a buffer, which it then compresses using zlib and forwards to the transport layer in the format shown in Figure 3.3.

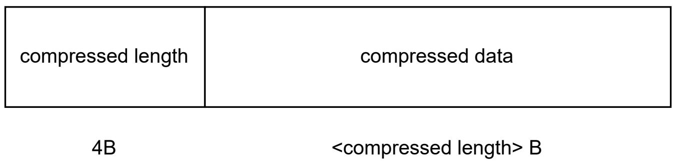

Figure 3.3: The format of the compressed data.

Even if the application sends its data in small chunks, the buffering implemented by TCP will bundle them into larger segments before sending them into the network, so that problems associated with small segments, like wasting bandwidth on a larger number of segment headers, will be avoided.

However, since compression tends to perform better (achieving a higher compression ratio) when the input data is large, the library inserts the application data into a buffer, and when enough data has been stored, its contents are compressed and passed to TCP. We summarize the sending process in the flowchart in Figure 3.4.

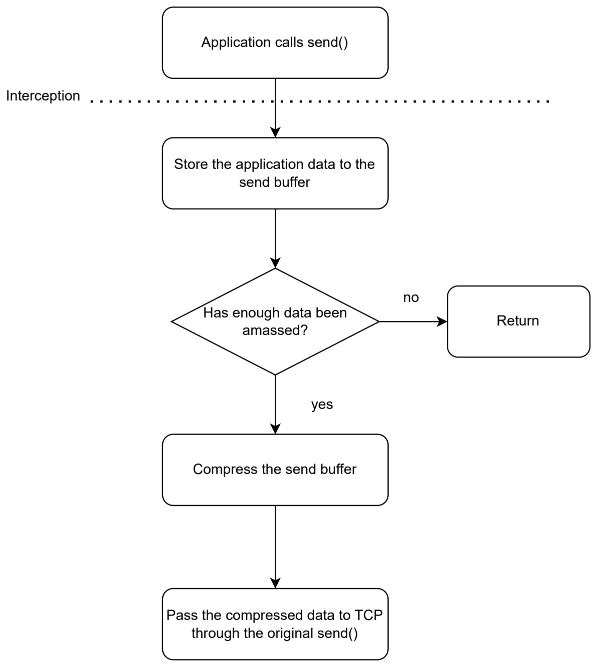
Application calls send()
Interception
Store the application data to the send buffer
Has enough data been amassed?
no
Return
yes
Compress the send buffer
Pass the compressed data to TCP through the original send()

Figure 3.4: The functionality of the sending side.

3.2.2 Receiving side

When the application requests `len` bytes of data using `recv()`, the receiving side receives compressed data from TCP in the format shown in Figure 3.3, which it then decompresses using zlib and uses it to deliver the `len` bytes. The whole process for receiving data is summarized in Figure 3.5.

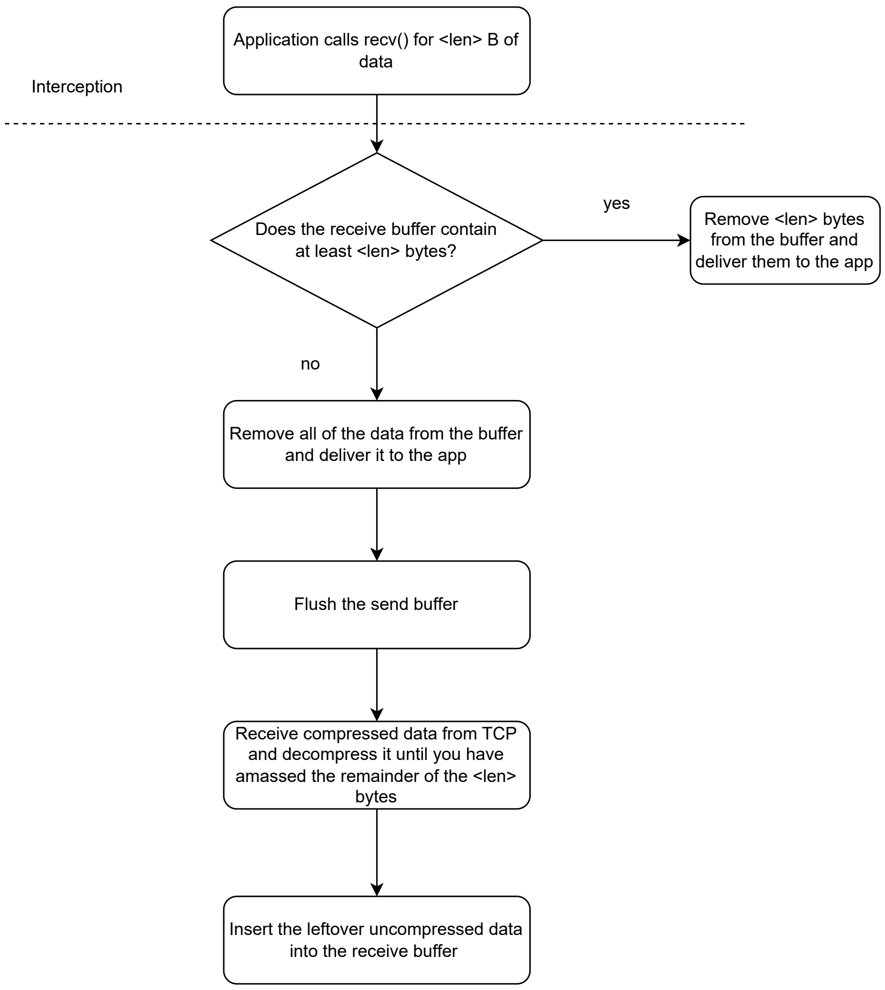
Application calls recv() for <len> B of data
Interception
Does the receive buffer contain at least <len> bytes?
yes
Remove <len> bytes from the buffer and deliver them to the app
no
Remove all of the data from the buffer and deliver it to the app
Flush the send buffer
Receive compressed data from TCP and decompress it until you have amassed the remainder of the <len> bytes
Insert the leftover uncompressed data into the receive buffer

Figure 3.5: The functionality of the receiving side.

Because the decompressed data might be more than what the application needs at the moment, the leftovers should be stored in a buffer so that they can be delivered to the application at a later `recv()` call. At every call, this receive buffer should be checked so that any data stored there can be retrieved to be returned to the application. If more data than what was stored in the buffer was requested, the library should receive more compressed data from TCP, decompress it , and repeat the process.

Also, a `send()` call for a small amount of data followed by a `recv()` call will cause the application to block forever, as the small data will never be sent due to the send buffering mentioned in Section 3.2.1, so the recipient application will never send its response. The solution to this problem is to flush the send buffer (by compressing the stored data and passing it to TCP even if it is a small amount) if the data in the receive buffer is not sufficient to serve the `recv()` call.

Chapter 4

Implementation

In this chapter, we discuss the various aspects of our C implementation. More specifically, how exactly the socket library calls are intercepted, the data structures that we used and the corresponding initialization/cleanup processes, and the finer details of the sending and receiving sides which were already discussed in Chapter 3.

4.1 Dynamic Library Interception

As mentioned in Section 3.1, the library doesn't offer an API that applications can use, instead it intercepts calls made to the libc socket API. In this section, we will describe exactly the steps we followed to achieve this.

For every socket API function that our implementation intercepts by implementing a function with the same name, after our own added functionality is finished, we must call the original function in order for the application to work as intended. The function `dlsym` from `<dlfcn.h>` is used to implement this behavior as follows:

```
typedef T0 (*orig_foo_t)(T1 arg1, T2 arg2, ..., TN argN);

T0 foo(T1 arg1, T2 arg2, ..., TN argN) {
    orig_foo_t orig_foo = (orig_foo_t)dlsym(RTLD_NEXT, "foo");

    added_foo_functionality(arg1, arg2, ..., argN);

    return orig_foo(arg1, arg2, ..., argN);
}
```

When dlsym is called using RTLD_NEXT as the first argument, it returns a function pointer to the next function named foo in the dynamic library stack (which comprises all the libraries that were dynamically linked at the start of the execution). As seen below, our library will be loaded first, so the next foo in the stack is the original.

To run an application using our library, we first compiled our implementation as a shared library using the following command:

```
gcc -DBUF_SIZE=204800 -shared -fPIC comp_tcp_lib.c -o comp_tcp_lib.so -ldl -lz
```

After that, the library's compression functionality can be used with any application that uses the libc socket API to communicate through TCP, by running the application in this manner (supposing that the compiled library is in the working directory):

```
LD_PRELOAD=$PWD/comp_tcp_lib.so <application-command>
```

This command loads the library before running the application, so the functions that have the same names as those in libc are higher in the dynamic library stack and will be called instead.

4.2 Compression Method

We opted to use the zlib lossless data compression library to implement the compression/decompression functionality of our library, because it is a mature library that is performant and easy to use. We will describe the way we used it in the following sections.

4.3 Data Structures

Provided the maximum number MAX_SOCKETS of sockets that will be created by the application is known, the data structures we used in our implementation are the following (Bytef and uLong are types provided by zlib and correspond to unsigned char and unsigned long respectively):

```
Bytef *send_buf[MAX_SOCKETS] = {NULL};
Bytef *send_comp_buf[MAX_SOCKETS] = {NULL};
Bytef *recv_comp_buf[MAX_SOCKETS] = {NULL};
Bytef *recv_buf[MAX_SOCKETS] = {NULL};
```

```
uLong send_len[MAX_SOCKETS] = {0};
uLong recv_pos[MAX_SOCKETS] = {0}, recv_len[MAX_SOCKETS] = {0};
```

Each of the above data structures is an array of pointers to byte buffers, or an array of length/position information associated with one of the buffers. They are indexed by the socket file descriptors returned by the original `socket()` or `accept()`, so each socket has a set of pointers to its own buffers. The reason for this is that an application might use a number of sockets concurrently from different threads. This way, each thread accesses completely separate memory areas from the others, so there is no risk of thread interference.

The byte array `send_buf[sockfd]` is the send buffer for the socket `sockfd` (as was mentioned in Section 3.2.1), that amasses application data until there is enough for efficient compression. The size of the array, which is also the amount of data required before compression takes place is `BUF_SIZE`. The array is filled from left to right, and the number of filled positions is stored in `send_len[sockfd]`.

The array `send_comp_buf[sockfd]` stores the compressed data that will be forwarded to TCP, while `recv_comp_buf[sockfd]` stores the compressed data that is received from TCP until it is decompressed. The size of this array is `compressBound(BUF_SIZE)` where `compressBound()` is a `zlib` function that computes the maximum size of the data that results from the compression of some amount of data.

Finally, `recv_buf[sockfd]` stores the decompressed data until the application requests it (as mentioned in Section 3.2.2). The size of this array is also `BUF_SIZE` because that is the maximum amount of data that will result from decompression. Once again, the array is filled from left to right, with the number of filled positions being stored in `recv_len[sockfd]`. The data is also consumed from left to right, so the number of consumed positions is also stored in `recv_pos[sockfd]`. These two variables are used to calculate how much data is left in the receive buffer.

4.4 Initialization

Each time the application calls `socket()`, our version is called instead. It first calls the original function to obtain a socket file descriptor. Then it checks whether the desired socket is a TCP socket or not, that is if `domain = AF\_INET` and `type = SOCK\_STREAM` from the provided arguments. If it is, the arrays introduced in Section 4.3 are allocated (using `malloc()`) and are associated with that socket file descriptor. In any case, the function returns the file descriptor to the application. The version of `accept()` we implemented behaves in a similar manner.

4.5 Sending side

Any application calls to `send()` are intercepted and our version is called instead. The provided `sockfd` is used to access the correct data structures.

The compression of the send buffer is performed using the `compress2()` function from zlib, which takes a data buffer and returns a compressed data buffer in `send_comp_buf[sockfd]`, using the default compression mode. Also, the compressed data are passed to TCP using the original send function in the format shown in Figure 3.3.

We will now discuss the buffering logic for the sending side. Supposing that the application wants to send `len` bytes of data from `buf`, the implementation attempts to insert it into `send_buf[sockfd]`. If `send_buf[sockfd]` does not fill up completely, there is not enough data for efficient compression, so our `send()` just returns `len` without forwarding anything to TCP. However, if the buffer fills up, it is compressed and then passed to TCP.

In case the implementation can not insert all of the `len` bytes into the send buffer, it separates them into groups of `BUF_SIZE` bytes, each of which it then compresses and forwards to TCP. Please note that the data is taken directly from `buf` instead of being put into `send_buf[sockfd]` first, in order to avoid wasting time on unnecessary copying.

If some bytes still remain, or there were not enough to create any of the aforementioned groups in the first place, they are just inserted into `send_buf[sockfd]` and will be utilized in a future `send()` call.

In every case, our version of `send()` returns the size of application data `len`, so it complies with the libc socket API.

The flowchart in Figure 4.1 summarizes the above functionality. We provide a simplified version of the actual buffering logic implementation in Appendix A.1.1.

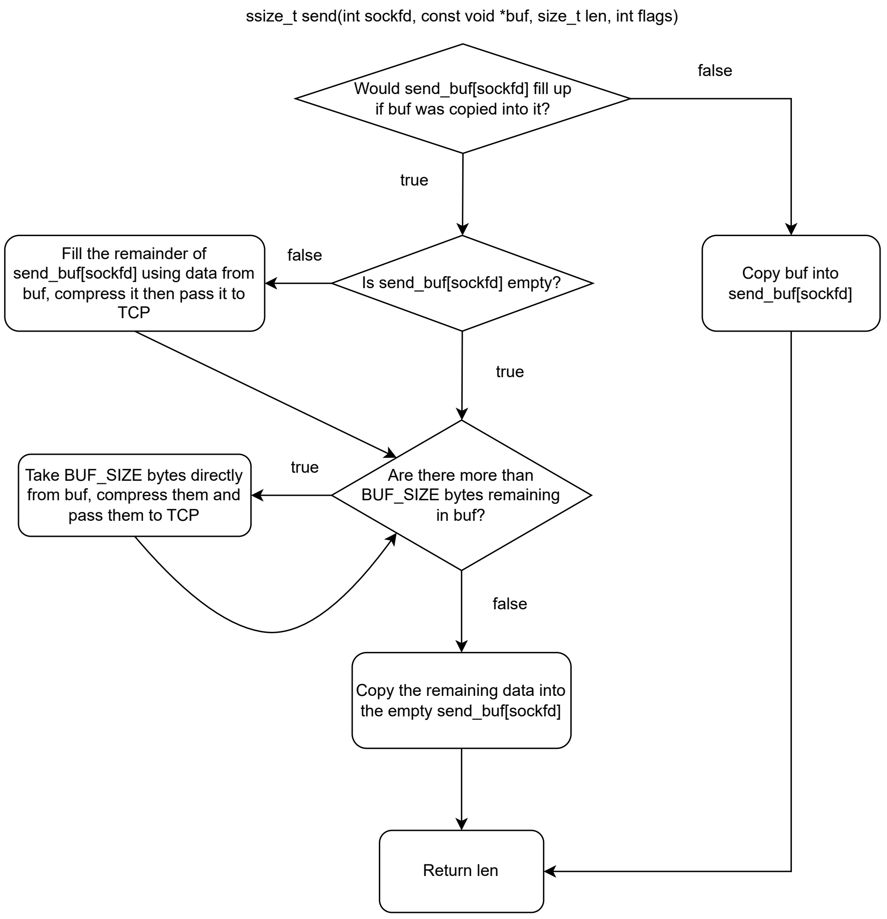
ssize_t send(int sockfd, const void *buf, size_t len, int flags)
Would send_buf[sockfd] fill up if buf was copied into it?
false
true
Copy buf into send_buf[sockfd]
Is send_buf[sockfd] empty?
false
Fill the remainder of send_buf[sockfd] using data from buf, compress it then pass it to TCP
true
Are there more than BUF_SIZE bytes remaining in buf?
true
Take BUF_SIZE bytes directly from buf, compress them and pass them to TCP
false
Copy the remaining data into the empty send_buf[sockfd]
Return len

Figure 4.1: The sending side implementation buffering logic.

4.6 Receiving side

Similarly to the sending side, the application calls to `recv()` are intercepted, our version is called instead, and the provided `sockfd` is used to access the correct data structures.

Compressed buffers are received from TCP using the original `recv()` and decompressed in order to be delivered to the application. The decompression is done using `zlib`'s `uncompress()` function, which takes a `zlib` compressed data buffer (`recv_comp_buf[sockfd]` in our case), and outputs the original data buffer.

4.6.1 Receiving compressed data from TCP

The interfacing with TCP is more complex than that in `send()`. We have encapsulated this functionality in the helper function `_recv_comp_data()`, which fills `recv_comp_buf[sockfd`
with a compressed buffer.

First, the original `recv()` is called in order to read the length header shown in Figure 3.3.

If at least some bytes have been delivered to the application, it is called in non-blocking mode. The reason for this is that fewer than expected bytes might have been sent to the application and waiting for the exact number it requested will lead to it blocking, but at the same time some bytes must be delivered. If the read fails with `errno = EAGAIN` or `EWOULDBLOCK`, the implementation will stop trying to fetch more compressed data for the remainder of this call to our `recv()`.

If the other side closed the connection and 0 was returned, our `recv()` is notified of that fact because it will receive a compressed buffer of length 0 and will stop fetching data, similarly to the non-blocking case.

If the length header was received successfully, the original `recv()` is called until the `length` bytes of the compressed buffer have been amassed in `recv_comp_buf[sockfd]`. Afterwards `_recv_comp_data()` returns and our `recv()` goes on to decompress the buffer.

The whole process is summarized in the flowchart in Figure 4.2.

4.6.2 Flushing

As mentioned in Section 3.2.2, flushing the send buffer is necessary whenever the application waits for data to be delivered from the network, else the application will block indefinitely. The flushing mechanism is simple: a helper `_flush()` function is called, which compresses the data in `send_buf[sockfd]` even if it is less than `BUF_SIZE` bytes long, and passes the compressed buffer to TCP.

4.6.3 Buffering logic

Supposing that the application requests that `len` bytes are delivered into `buf` by calling `recv()`, the implementation attempts to service the request using the contents of `recv_buf[sockfd]`. If it contains enough data, `len` bytes are copied from it to `buf` and the function returns `len`.

In case the bytes in `recv_buf[sockfd]` are less than `len`, all of its bytes are copied into `buf`, and `send_buf[sockfd]` is flushed. Afterwards, the implementation receives compressed buffers from TCP, which it then decompresses and copies directly into `buf`. This process continues as long as the application needs more than `BUF_SIZE` bytes to be delivered. That is because the maximum size for a decompressed buffer is `BUF_SIZE`, so if the application needs less than that, the remainder of the buffer has to be handled somehow.

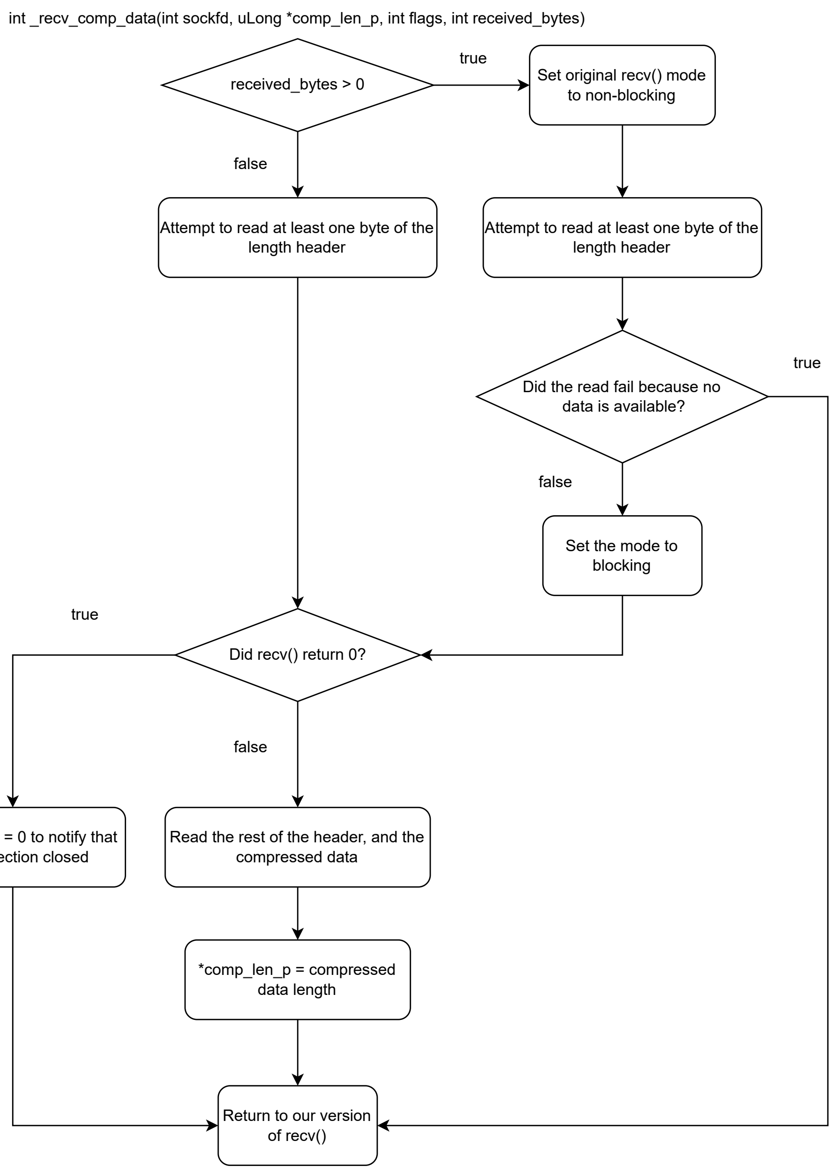
int _recv_comp_data(int sockfd, uLong *comp_len_p, int flags, int received_bytes)
received_bytes > 0
true
Set original recv() mode to non-blocking
false
Attempt to read at least one byte of the length header
Attempt to read at least one byte of the length header
Did the read fail because no data is available?
true
false
Set the mode to blocking
true
Did recv() return 0?
false
*comp_len_p = 0 to notify that the connection closed
Read the rest of the header, and the compressed data
*comp_len_p = compressed data length
Return to our version of recv()

Figure 4.2: The helper function for receiving compressed buffers from TCP.

To deliver the remaining bytes to the application, the implementation receives one final compressed buffer from TCP, which it then decompresses. Whatever bytes the application needs are copied from the decompressed buffer to `buf`, and the rest are copied to `recv_buf[sockfd]` for future use.

In any case, our `recv()` function returns the size of the data delivered to the application (which might be less than `len`), so it respects the libc socket API. Also, when the connection has been closed and no more data is left to be delivered, it returns 0 to notify the application that EOF has been reached, just as the original `recv()`.

Once again, the flowchart in Figure 4.3 that summarizes the above functionality, and Appendix A.1.2 provides a simplified version of the actual buffering logic implementation.

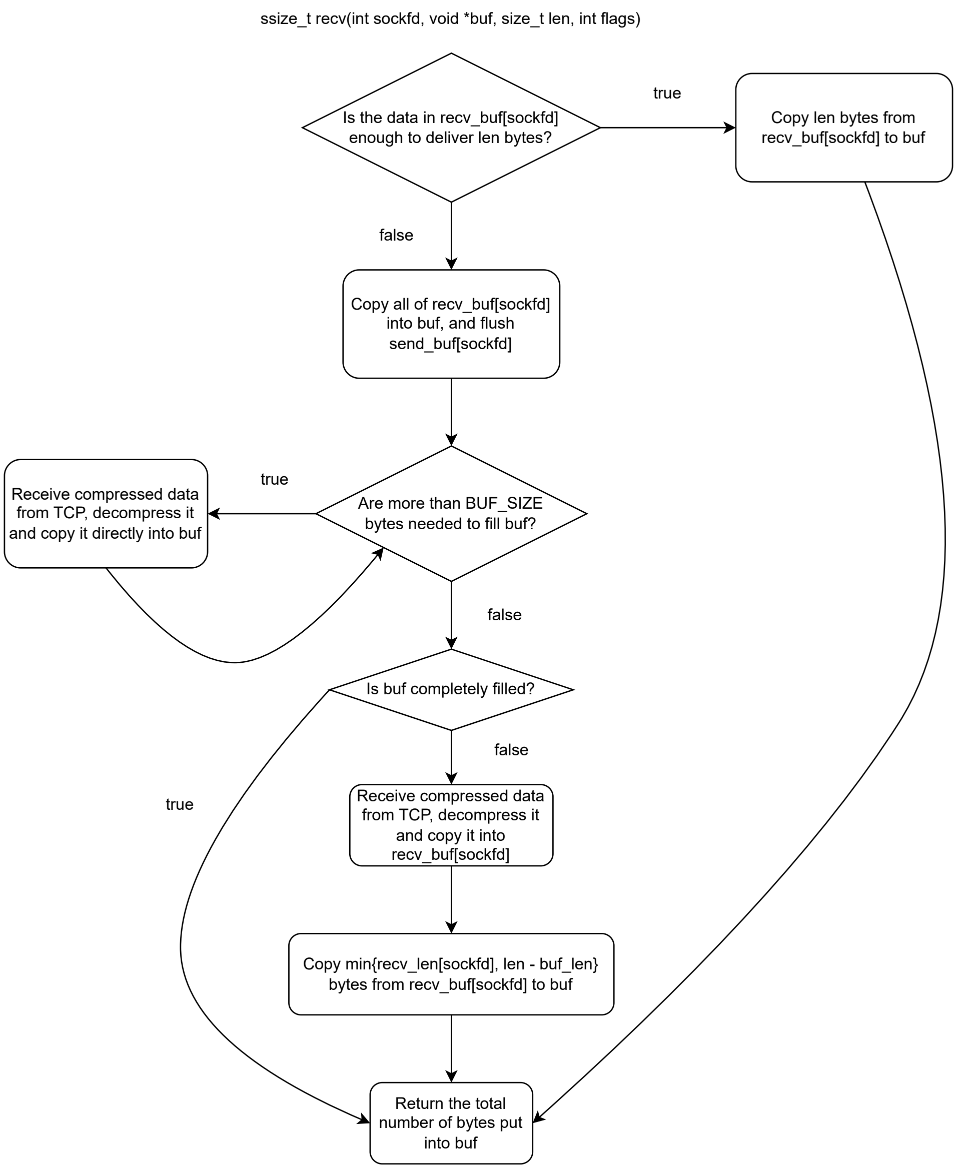
ssize_t recv(int sockfd, void *buf, size_t len, int flags)
Is the data in recv_buf[sockfd] enough to deliver len bytes?
true
Copy len bytes from recv_buf[sockfd] to buf
false
Copy all of recv_buf[sockfd] into buf, and flush send_buf[sockfd]
Are more than BUF_SIZE bytes needed to fill buf?
true
Receive compressed data from TCP, decompress it and copy it directly into buf
false
Is buf completely filled?
false
true
Receive compressed data from TCP, decompress it and copy it into recv_buf[sockfd]
Copy min{recv_len[sockfd], len - buf_len} bytes from recv_buf[sockfd] to buf
Return the total number of bytes put into buf

Figure 4.3: The receiving side implementation buffering logic.

4.7 Cleanup

Once again, when the application calls `close()`, our version is called instead. It checks if the file descriptor corresponds to a TCP socket by checking if the library data structures for that file descriptor have been created or not. If it does, the send buffer is flushed (as in `recv()`), and then the data structures associated with the file descriptor are freed. Finally, the original `close()` is called on the file descriptor.

Chapter 5

Evaluation

In this chapter, we describe the experimental setup we used to validate the correctness of our implementation, as well as to evaluate the efficiency and resource utilization of the compression process and the performance of our implementation versus that of plain TCP in different network conditions. Also, we present and discuss the results of our experiments.

5.1 System specifications

We performed all of the experiments described in this chapter on a desktop computer with the specifications shown in Table 5.1.

OS	Ubuntu 20.04.5 LTS x86_64
Kernel	5.4.0-139-generic
CPU	Intel i5-7500 (4) @ 3.800GHz
Memory	2 x 8GB
zlib	1.2.11
Mininet	2.3.0.dev6

Table 5.1: The specifications of the machine we used to run our experiments.

5.2 Benchmarks

In this section, we will discuss the characteristics of the files (datasets) we used to test our implementation. We obtained some of these files from the Canterbury Corpus [17], a file set specifically developed for testing lossless compression algorithms. The files relevant to this chapter are shown in Table 5.2.

Descriptor	Size (MB)	Contents	Corpus name
bin	12.8	Executable	-
non-fiction	0.6	Non-fiction book (troff format)	book2
E.coli	4.4	Complete genome of the E. Coli bacterium (txt)	E.coli
xls	1.0	Excel spreadsheet	kennedy.xls
src	2.4	C# source code	-
fiction	4.3	Fiction book (txt)	-
cia guide	2.4	The CIA world fact book	world192.txt
png	10.5	An image in the png format	-
mp4	29.2	A video in the mp4 format	-
zip	1.3	A compressed archive in the zip format	-
pdf	79.5	A pdf document	-
mp3	14.1	An audio file in the mp3 format	-
jpg	0.8	An image in the jpg format	-
megabook	32.4	Concatenation of multiple literary works (txt)	-
enwik8	95.4	First 10^8 bytes of the English Wikipedia dump on 3/3/2006	-
enwik9	953.7	First 10^9 bytes of the English Wikipedia dump on 3/3/2006	-

Table 5.2: The files used in the evaluation of our library.

5.3 Experimental setup

We wanted to perform experiments in order to:

- Validate the correctness of the implementation.
- Evaluate the performance of the compression method we used.
- Quantify system resource utilization on a single communication node.
- Compare the performance of plain TCP with that of our library in different network conditions.

5.3.1 Basic setup

We built the main experimental platform around a simple one-way transmission scenario: A client application (which we will refer to as `oneway_client`) reads a file from the disk in chunks, sends the data in the file, using our compression library, to a server application (which we will refer to as `oneway_server`), which in turn writes the received chunks to the disk.

To validate our implementation, we run the experiment for each file mentioned in Section 5.2 for various values of `BUF_SIZE`, then used `diff` to compare the file read by `oneway_client` with that written by `oneway_server`. This way, we made sure that the implementation runs correctly for a variety of cases in the buffering logic. We also used another client-server pair, `echo_client` and `echo_server`, in which the client read the file in chunks like before, but after every chunk it sent, it waited to receive that same chunk from the server before writing it to the disk. We did that in order to make certain the implementation worked correctly when nodes act as both senders and receivers.

5.3.2 Network emulation setup

The above setup was also used for simple performance measurements. However, in order to reliably evaluate our implementation for different network conditions in a controlled way, we had to use the network emulator Mininet.

Our first attempt at creating such a setup was to run the aforementioned client-server pair inside Mininet, where we would be able to create the desired conditions by setting link parameters such as bandwidth, latency, etc. The issue with this approach was that the client (which compresses data), the server (which decompresses data), and Mininet all run on the same computer, so they compete for system resources. Therefore, any resource utilization measurements would be invalid.

Another idea was to run those three programs in separate nodes. Besides the difficulty in implementing this solution, due to programs running inside Mininet not being able to communicate easily with external programs in the same host through their network interfaces, and the performance degradation because of the addition of extra recipient programs between the original server-client pair because time inside the Mininet simulation is slower than time in real life, this solution would provide time measurements for the whole of the client-Mininet-server system that are much higher than what they would be on a real network.

In the end, we decided that `oneway_client`, Mininet and `oneway_server` must run sequentially in order to satisfy all of the experiment requirements. We achieved that by modifying the library so that it reads from, or writes to logs containing timestamps and compressed buffers, instead of forwarding or receiving data from TCP. This way, while the components of the system do not run at the same time, they can read the timestamps and use them to simulate the delays in their execution if they had to wait for data from another component. An added benefit to this method is that the setup could be run on just one machine.

The final setup we used is shown in Figure 5.1. We will now describe the latency breakdown steps shown in the figure. Each step corresponds to a step of the actual concurrent communication between `oneway_client` and `oneway_server` that we started with.

Step 1 - Compression and sending

The original client `oneway_client` reads the file from the disk in chunks, that are passed to our implementation, which creates two logs, one that contains <compression buffer size, compressed data> pairs which we will refer to as `buf_log` from now on, and another one that contains <compression end timestamp, compression buffer size> pairs, which we will call `comp_ts_log`.

Step 2 - Transmission through the emulated network

A new client-server pair `mininet_client` and `mininet_server` is run in two virtual hosts of a `Mininet` network that emulates the desired network conditions.

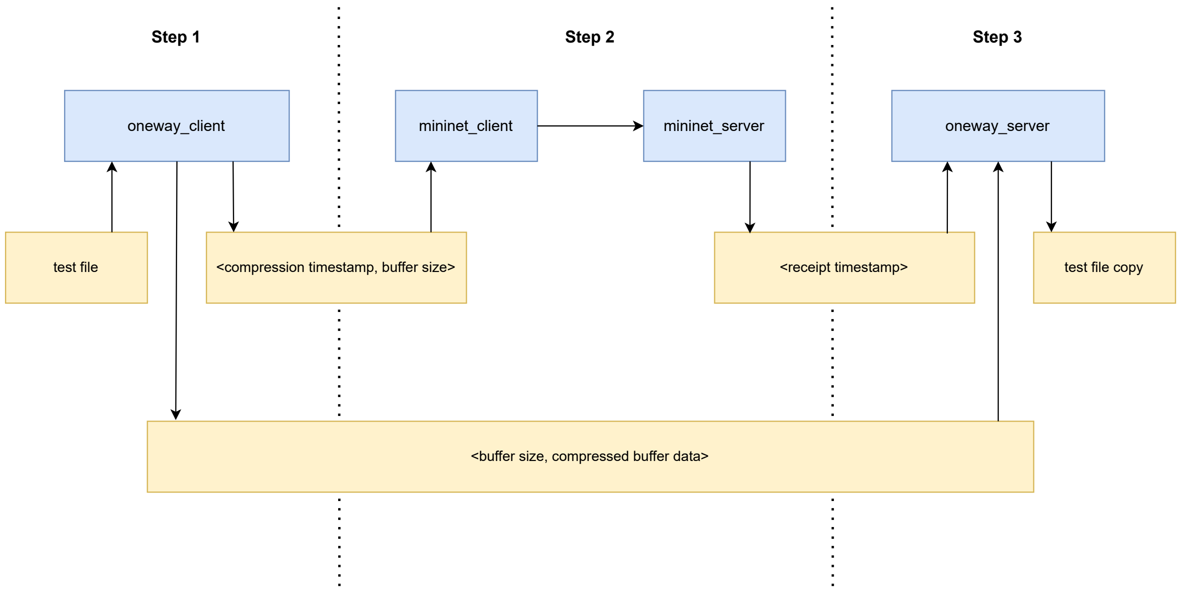

Figure 5.1: The experimental setup. Blue denotes a program while yellow denotes a file.

The client reads `comp_ts_log`, and uses it to send byte buffers filled with zeros which have the same size as the compressed buffers generated in Step 1, after waiting for the difference between two timestamps in the log `comp_ts_log[i] - comp_ts_log[i-1]`, which corresponds to the duration of the compression of that buffer, for each one. By doing that, the delays between the end of one compression and the beginning of another are reproduced just as they happened in `oneway_client`, ensuring that the behavior is close to that in the original setup, even though the task of compressing a buffer and sending it is split into two applications.

The server reads `comp_ts_log` in order to know the size of the buffers it will receive. Afterwards, each time it receives a compressed buffer, it writes the current timestamp in another log, named `recv_ts_log`.

Step 3 - Receipt and decompression

The original server application `oneway_server`, which received the file from `oneway_client` and wrote it to the disk is run next. My implementation starts by noting the current time and then reads the contents of `recv_ts_log`. Each time a chunk is requested by the application, the implementation waits until the time corresponding to the next timestamp in `recv_ts_log`. Then, it reads a compressed buffer from `buf_log` which it then decompresses and delivers to the application as is normally done.

Once the application closes the socket, the current time is noted, and then the difference `close_time - init_time` is computed. This is the quantity we take as the measurement of the total execution time for the system. This is valid because the waiting time written in `recv_ts_log` essentially contains the delays from the client-side disk reading, the compression of the buffers, and the traversal through the network.

This process was run 10 times for each set of network parameter values we wanted to test, with the final time measurements resulting from averaging the 10 measurements. The chunk size that `oneway_client` uses to read the file is 10240 B.

5.4 Compression efficiency evaluation

In the following subsections, we will provide experimental results plots for various metrics and the corresponding explanations. All of the X-Y plots use a logarithmic x-axis with base 2. Also, the values of each metric are the average of the measurements from 10 experiments to eliminate the statistical error. We omitted Mininet in this set of experiments because we only wanted to measure the compression algorithm's performance.

5.4.1 Compression ratio

First, we set `BUF_SIZE` to 102400 and then ran the experiment for files with different formats in order to better understand how well compression would work for each of them. As seen in Figure 5.2, the size of most of the files is not reduced after compression or it might even be increased (due to extra headers added by the compression algorithm). We expected that since those files are images, videos, zip archives, or pdf documents, all of which are already compressed.

On the other hand, most of the chunks of the text files and the executable had their size reduced by a factor of 2 or more, so the compression was meaningful in this case. This makes sense, as files in these formats are not already compressed and have a lot of redundant information which can lend itself to efficient compression. This is especially apparent in the case of the source code file, which contains text in a programming language where the syntax is strict leading to a larger degree of redundancy, where 75% of its chunks had their size reduced by a factor of more than 6.

So, we concluded that some commonly transmitted file formats like text files can be compressed successfully, therefore our approach could lead to an increase in performance in adverse network conditions as we hypothesized.

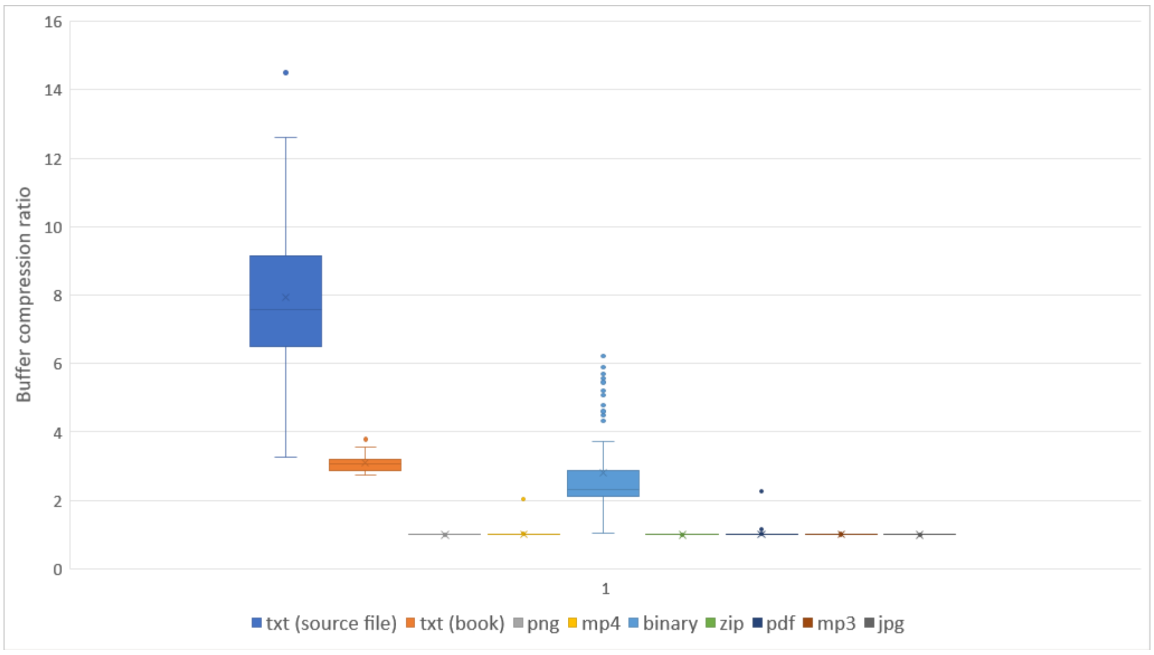

Figure 5.2: Boxplot of the compression ratio for files of different formats.

We calculated the mean compression ratio for a given execution of the above setup by averaging the compression ratios for each `send_buf[sockfd]` passed to `compress2()`. To compute the individual ratios, we just divided the uncompressed size by the compressed size. We repeated the above process for various files and `BUF_SIZE` values, leading to the plot in Figure 5.3.

As `BUF_SIZE` which is also the size of the compression unit increases, the compression ratio becomes larger until some limit value (after some point the increase in redundancy is negligible given that most files are probably homogenous in that respect). Most files reached their final mean compression ratio by `BUF_SIZE = 204800` so that is a good choice for a default `BUF_SIZE` value with respect to compression efficiency.

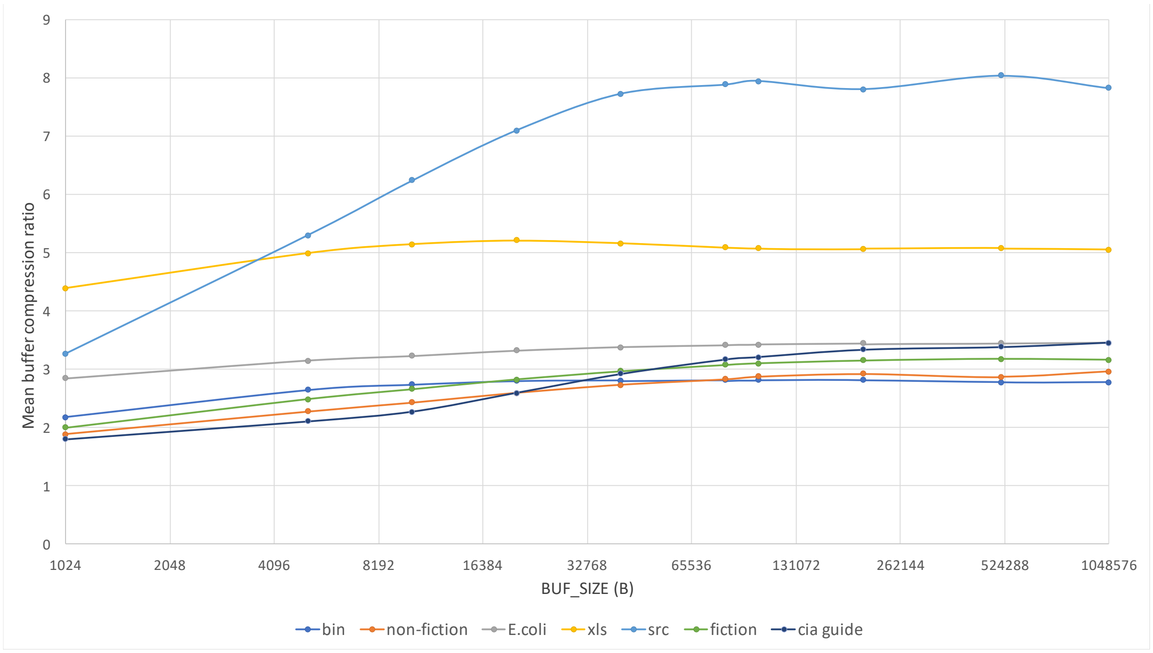

Figure 5.3: Plot of the mean compression ratio vs BUF_SIZE for various files.

5.4.2 Compression overhead

We used the file referred to as ”enwik8” in Section 5.2 because we wanted to use a larger file for performance testing. Also, we measured the time taken by `zlib`’s `compress2()` and `uncompress()` for each buffer, using `clock()`. Using those values, we calculated the total compression and decompression times (by adding them) and the mean compression and decompression time for a single buffer (by averaging them).

We plotted the total compression time in Figure 5.4. We expected it to decrease as `BUF_SIZE` increased, because fewer calls to `compress2()` would be made, which creates data structures for the DEFLATE algorithm each time it is called (as we found out by studying the source code), so fewer allocations and then deallocations are performed. This does happen until a certain point, however, afterwards the total compression time starts to increase again.

We suspected that this was due to the buffers growing too large to fit inside the cache leading to an increased cache miss rate and decreased performance. In order to test our hypothesis, we ran the client using `perf` and observed the values of the performance counters relevant to cache performance. Most of them did not change with the increase in buffer size, but we found that the value of `L1-dcache-load-misses` started to increase at the same point as the total compression time, so our hypothesis was confirmed. The L1-dcache miss rate is plotted in Figure 5.5.

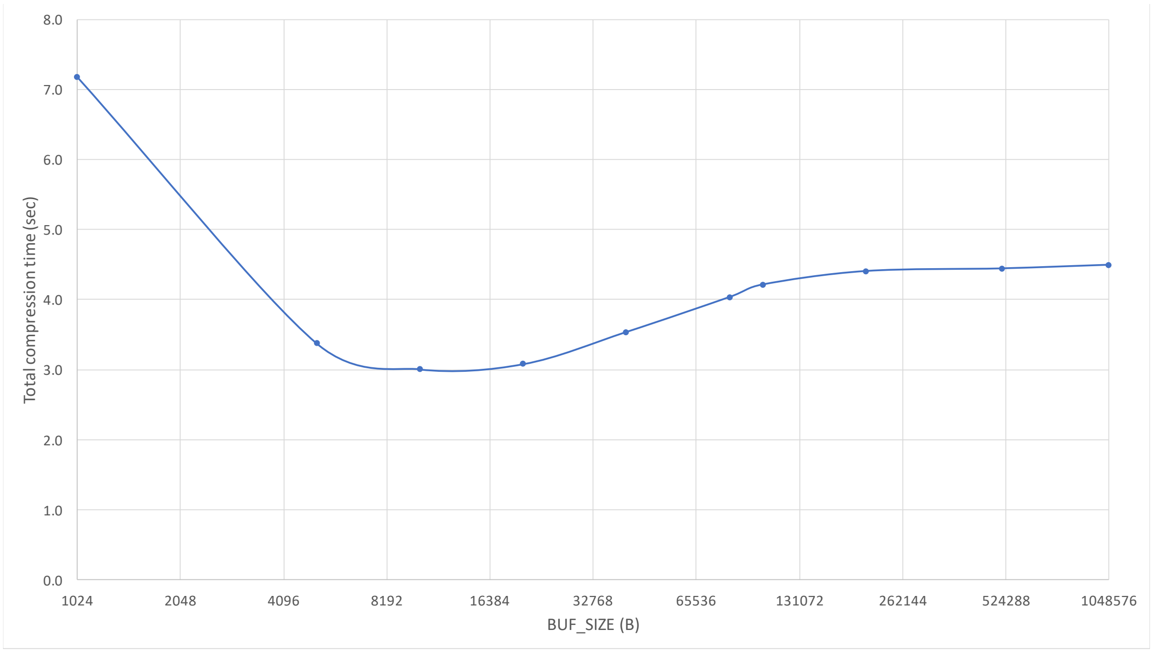

Figure 5.4: Plot of the total compression time vs BUF_SIZE.

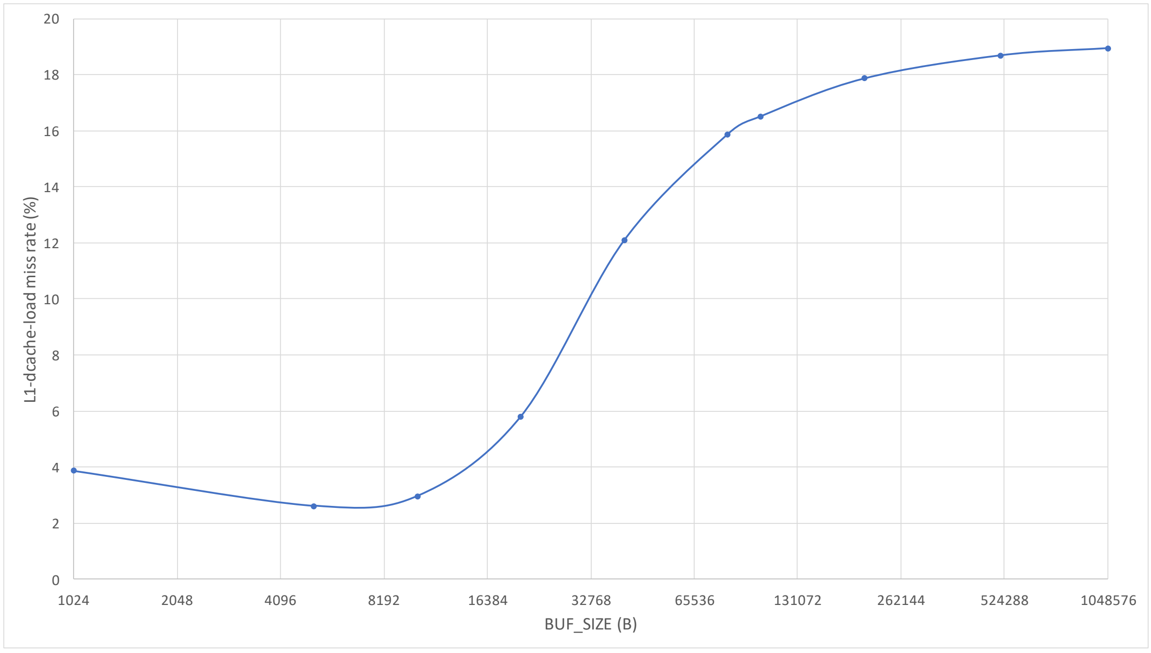

Figure 5.5: Plot of the L1-dcache miss rate (%) vs BUF_SIZE.

We have also plotted the mean compression time for a single buffer in Figure 5.6. It increases linearly with `BUF_SIZE` (the curve is exponential in a lin-log graph), which was expected as DEFLATE (which is used by `zlib`) is an $O(N)$ algorithm.

The total decompression time in Figure 5.7 only decreases as `BUF_SIZE` increases, as we expected. Also, the mean decompression time for a single buffer in Figure 5.8 increases linearly with `BUF_SIZE`, which is normal considering that the INFLATE algorithm also has

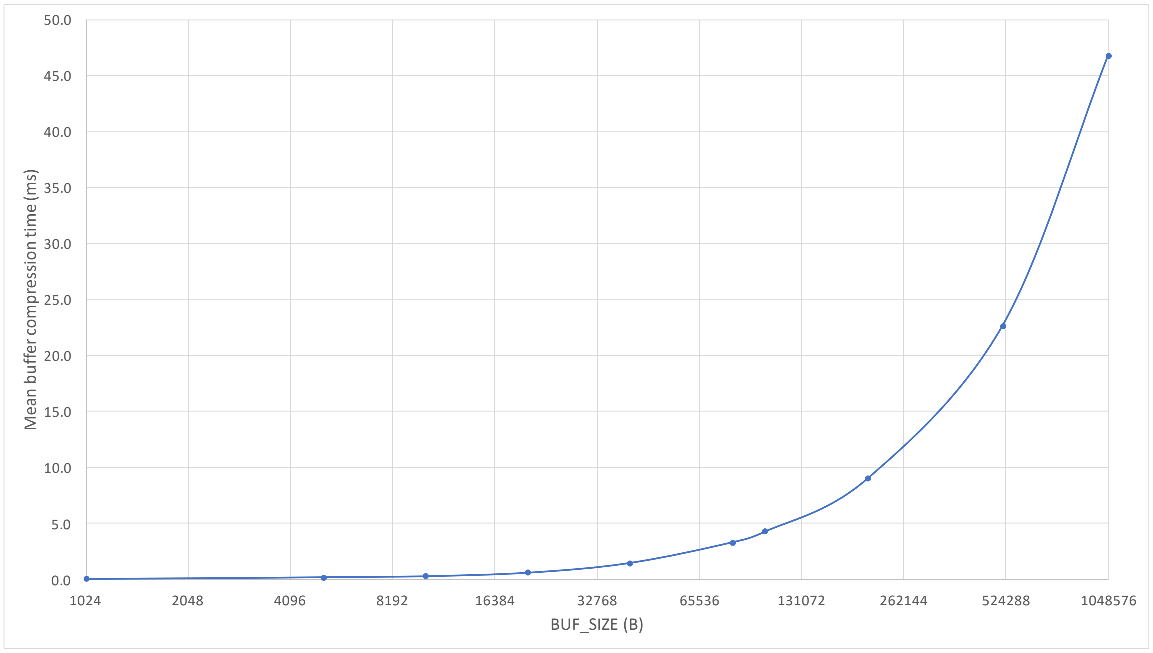

Figure 5.6: Plot of the mean buffer compression time vs BUF_SIZE.

linear time complexity.

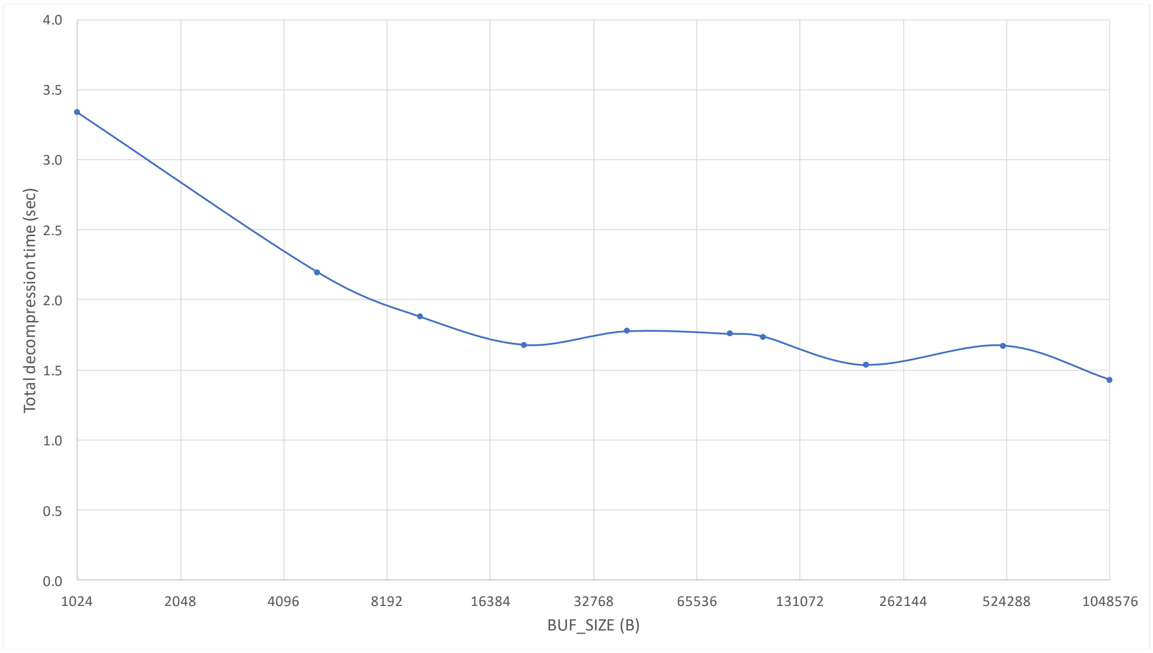

Figure 5.7: Plot of the total decompression time vs BUF_SIZE.

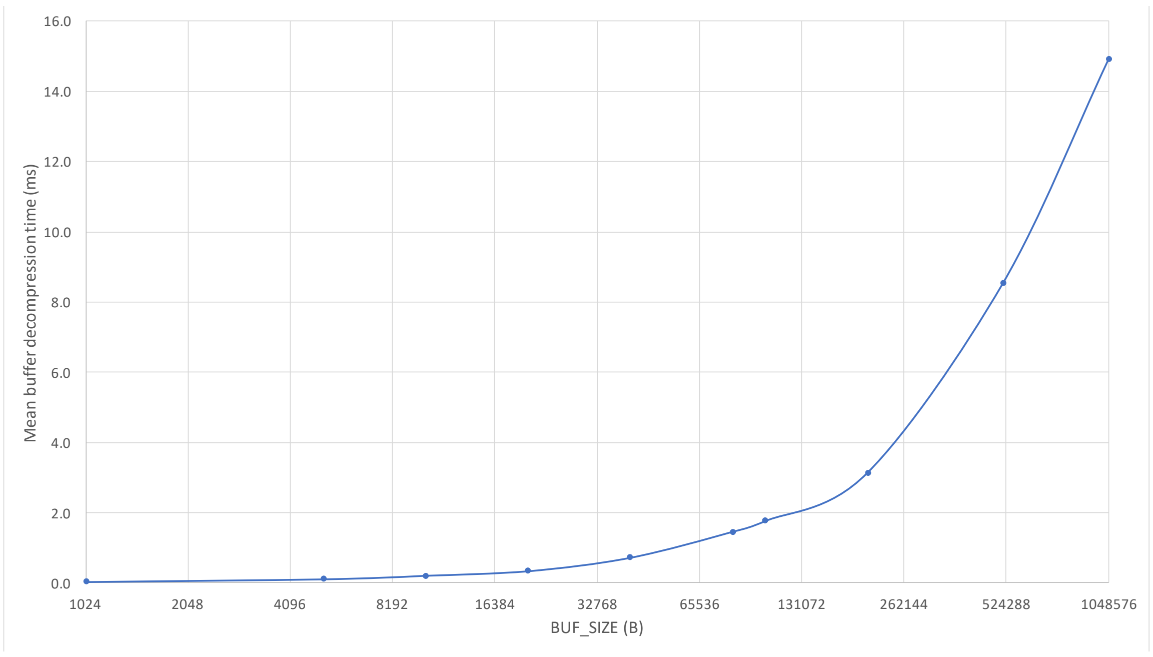

Figure 5.8: Plot of the mean buffer decompression time vs BUF_SIZE.

5.5 CPU overhead experiment

We measured the CPU load induced by our implementation by calculating the following quantity:

$$L(\%) = \frac{cputime}{realtime \cdot cores}\dot{1}00\% \tag{5.1}$$

Where `realtime` is the real time difference between the start of the initialization and the end of the cleanup of the library (the timestamps are computed using `gettimeofday()`) and `cputime` is the processor time difference between the same events (using `clock()`). Also, `cores` is the number of CPU cores in the system.

We started by investigating the relationship of the CPU percentage utilized by our implementation to `BUF_SIZE`, with the client and server communicating through the loopback interface. We plotted the CPU load for the sending side and the receiving side in Figure 5.9.

On the sending side, the percentage is $24-25\%$, independent of `BUF_SIZE`. Our system has 4 cores, so that is equivalent to using $96-100\%$ of one of them, which is expected since the implementation is single threaded and it spends most of the time doing CPU-bound compression, because sending data through loopback is very fast.

The receiving side has to wait for data to arrive so its CPU utilization is lower than that of the compression side.

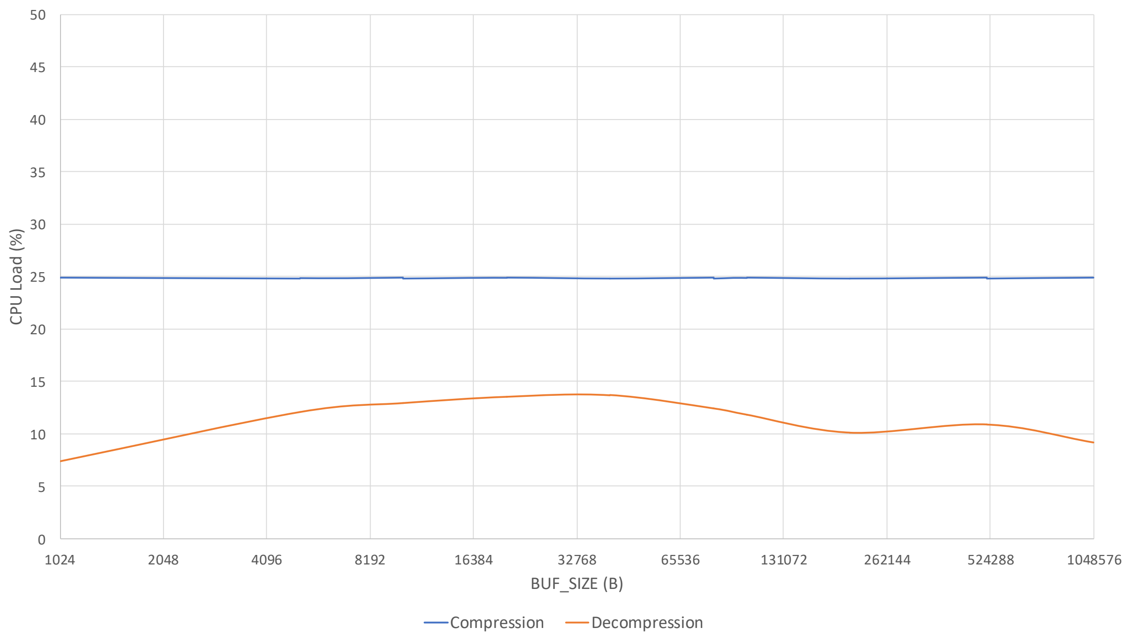

Figure 5.9: Plot of the CPU load for the compression and decompression vs BUF_SIZE.

One could conclude that the sending-side is CPU-bound because, during its execution, one CPU core is always utilized. However, we did not take into account the communication delays that are present when communicating through a network. We used the experimental setup we showed in Section 5.3 to measure the CPU utilization for both the sending and the receiving sides in various network conditions, using Mininet's bandwidth, latency, and packet loss rate parameters.

As shown in Figure 5.10, the CPU is utilized by both sides to a greater extent as the bandwidth increases. This is because if the bandwidth is small, the data is transmitted slowly and each side spends more time waiting for IO to finish.

When the latency increases, the CPU load is decreased for both sides as shown in Figure 5.11, because they once again spend more time waiting on IO operations. That is because TCP does not send more data if it has not yet received ACKs for the previously sent data, and these acknowledgments will arrive after at least one RTT has passed (provided that the packet or the acknowledgment was not lost). So if the latency is large enough, the RTT will be also large and the ACKs will take a long time to arrive, limiting TCPs effective bandwidth.

Finally, as shown in Figure 5.12, when the packet loss rate is large, the CPU load decreases. If packets are lost often in the network and TCP has to re-transmit them in order for them to reach the destination, the communication slows down and both the sender and the receiver spend more time waiting for IO operations to complete.

Based on all of the above, both the sending and the receiving sides of our implementation are actually IO-bound as expected for a communications library. Please note that we performed these experiments on a CPU that is stronger than the typical edge device CPU, in which case the CPU utilization percentage will be higher than the one we measured.

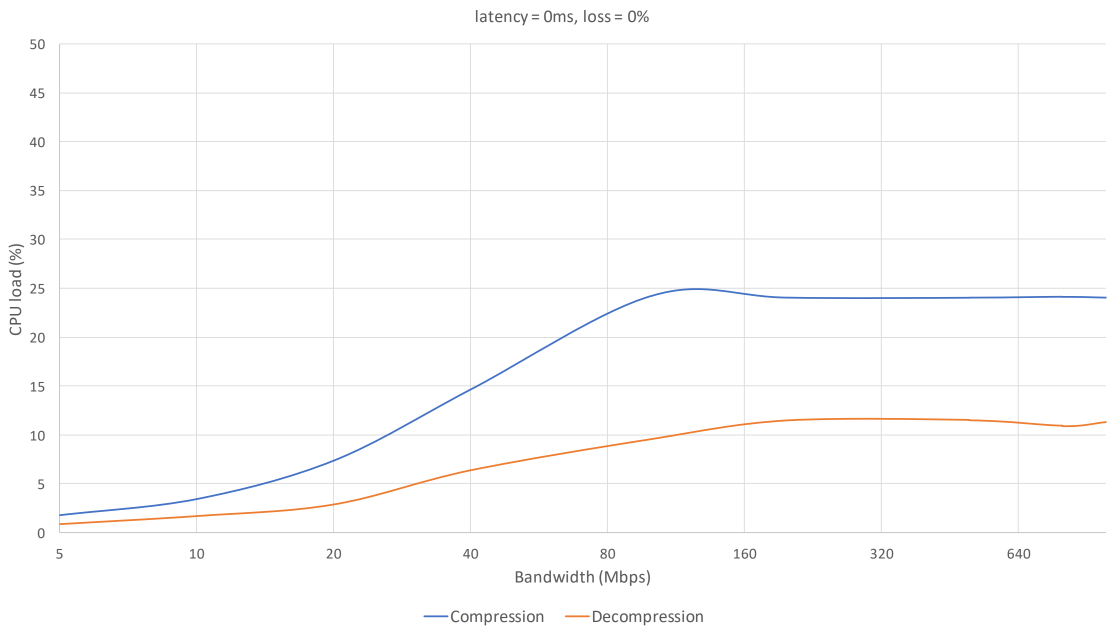

Figure 5.10: Plot of the CPU load vs bandwidth.

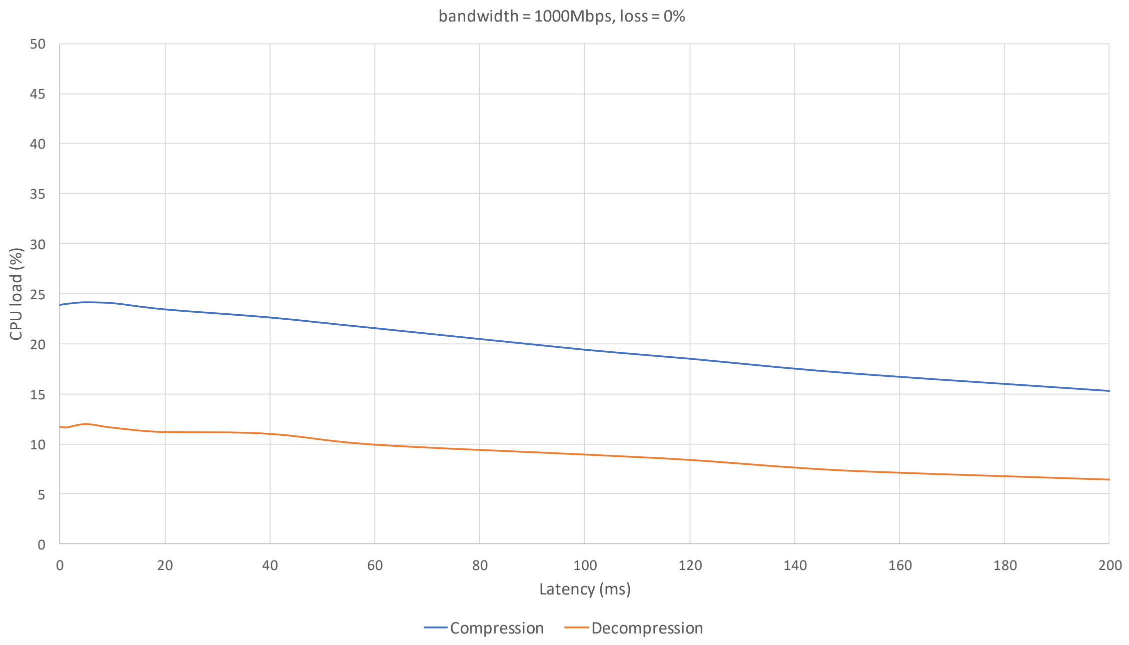

Figure 5.11: Plot of the CPU load vs latency.

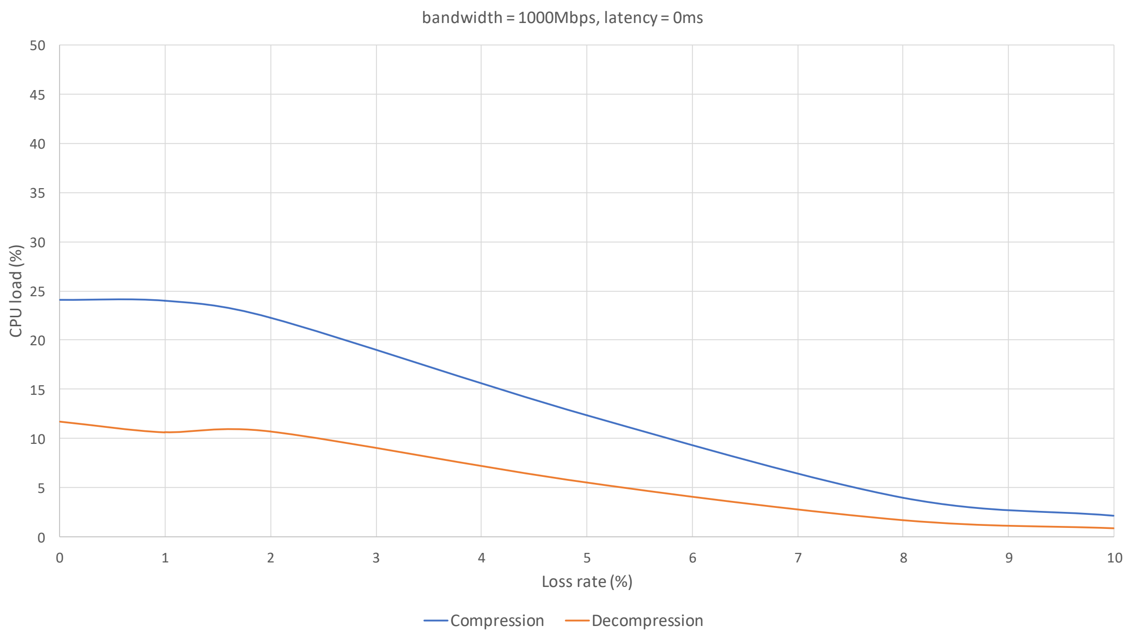

Figure 5.12: Plot of the CPU load vs packet loss rate.

5.6 Comparison between TCP and the library

In this section, we will compare the application execution time and effective bandwidth when using vanilla TCP with that of our compression library for different network conditions in order to experimentally validate our hypothesis that data compression can speed up communication when those conditions are adverse. The file we used is the same as that in Section 5.4.2, the chunk size was still 10240 B, and `BUF_SIZE` was set 204800 B.

5.6.1 Bandwidth

We started by investigating the effect of bandwidth on performance. To do that, we set the `Mininet` packet loss rate parameter to 0% and the delay parameter to 0 ms which are ideal values, and tried various bandwidth parameter values. Then, we used the setup outlined in Section 5.3 to produce the plots in Figures 5.13 and 5.14. The horizontal axis is in log scale with base 2.

When the bandwidth is small, the performance is considerably better using our library compared to TCP, as the data compression ensures that less data needs to be transmitted over the network. As the bandwidth increases, the difference between the performance becomes smaller and at some point, the performance of plain TCP becomes slightly better than that of the library. That happens because the network is fast enough to support the rate at which the application produces and sends the data, so the library just wastes time compressing it.

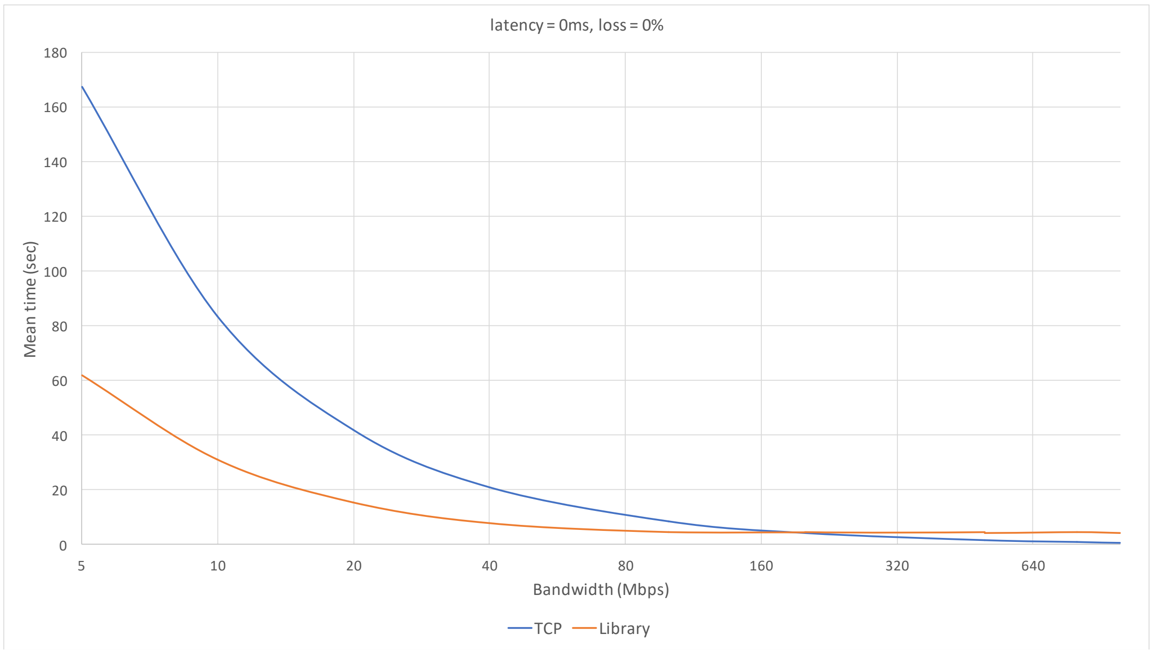

Figure 5.13: Execution time vs bandwidth comparison between TCP and the library.

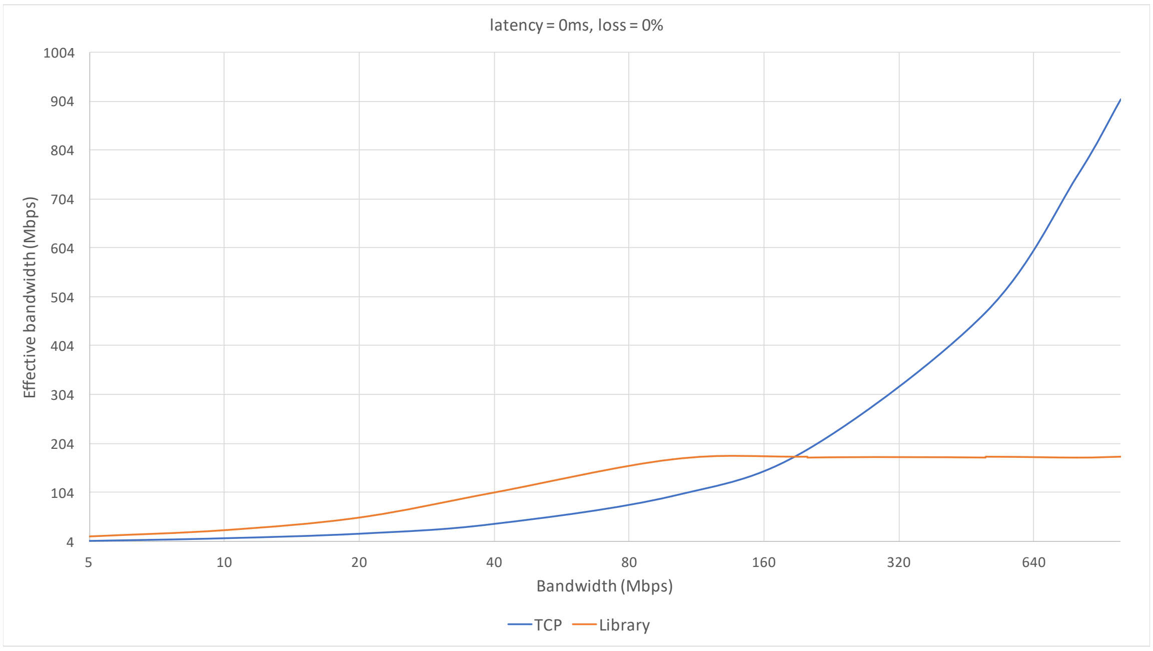

Figure 5.14: Effective bandwidth vs bandwidth comparison between TCP and the library.

5.6.2 Latency

To investigate the effect of latency, we set the packet loss rate parameter to 0% and the bandwidth parameter to 1000 Mbps, and performed a parameter scan on the delay parameter. The resulting plots are shown in Figures 5.15 and 5.16.

Initially, the latency is small and plain TCP performs better than the library. As it increases, the difference between them gets smaller and when the latency is large their positions are reversed.

Both our implementation and plain TCP face the issue discussed in Section 5.5, however in the case of our library, because the data is compressed and the same number of TCP segments contains more of it, the effective bandwidth is higher. However, compression is not worth it when the latency is not very high because the data (and the corresponding ACKs) would reach their destinations very quickly anyway, so time is wasted performing it.

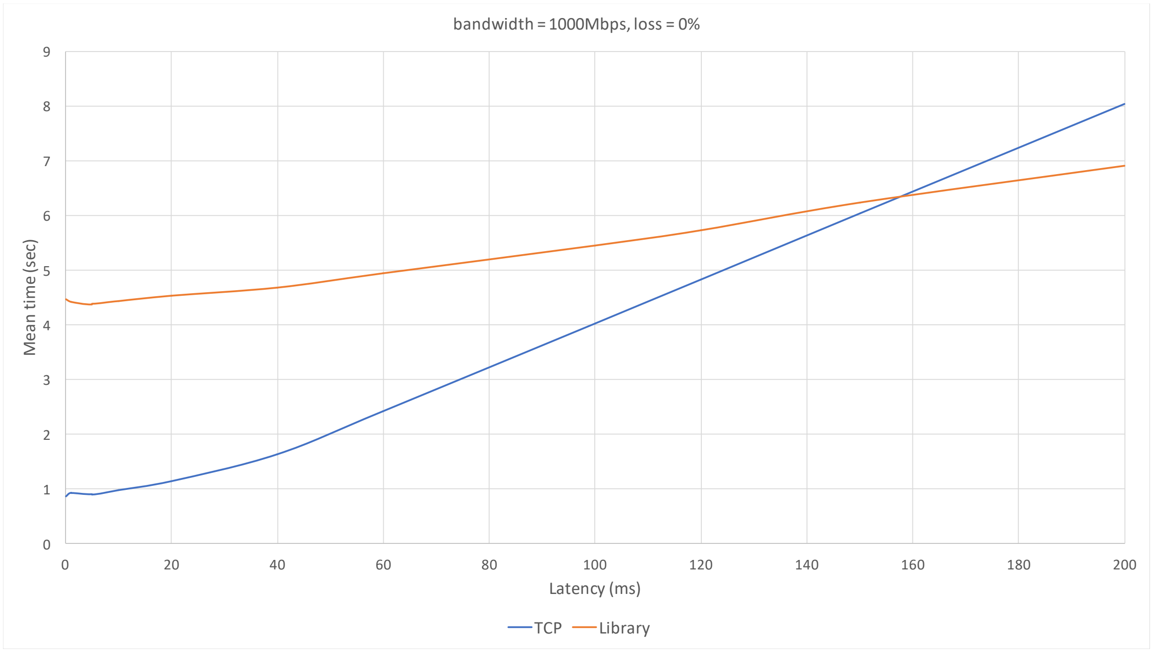

Figure 5.15: Execution time vs latency comparison between TCP and the library.

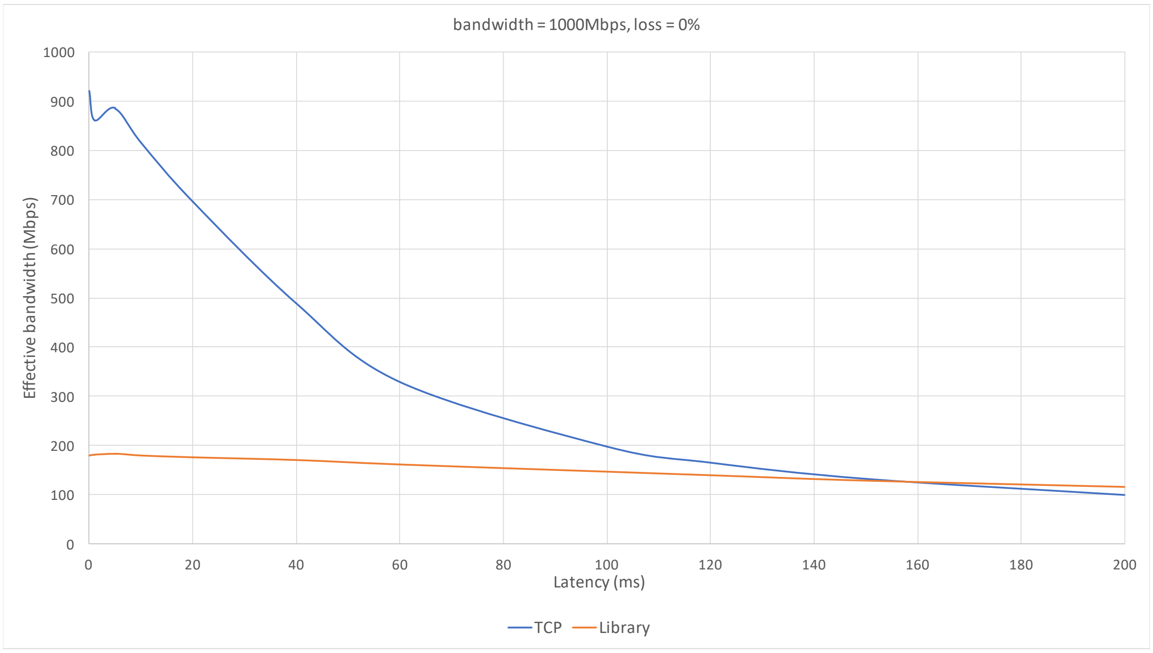

Figure 5.16: Effective bandwidth vs latency comparison between TCP and the library.

5.6.3 Packet loss rate

To investigate the effect of the packet loss rate, we set the bandwidth parameter to 1000 Mbps and the delay parameter to 0 ms and performed a parameter scan. The resulting plots are shown in Figures 5.17 and 5.18.

When the loss rate is small, plain TCP performs slightly better than the library, but with larger loss rates the library performs considerably better compared to TCP. With data compression, fewer data have to be sent so fewer TCP segments are created and transmitted. If fewer packets are transmitted, the probability of some of them being lost and having to be re-transmitted is smaller. So, when using the library fewer packets are lost and less time is wasted on re-transmissions compared to using plain TCP. As with the other parameters, when the loss rate is small, very few packets will be lost so the data compression just wastes time.

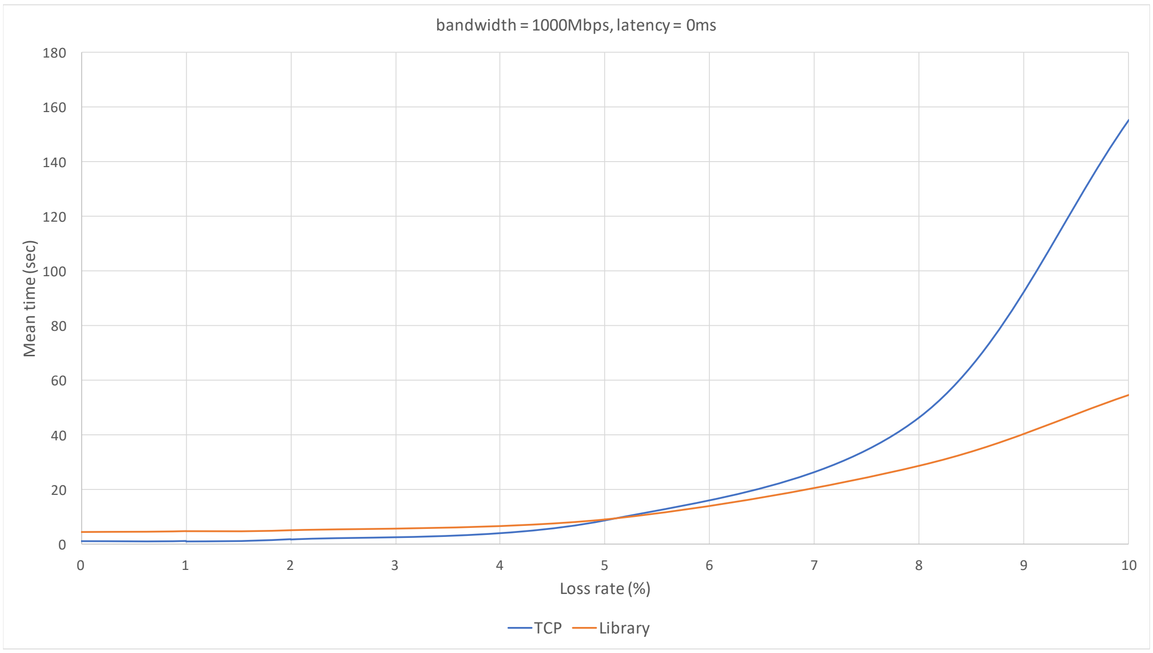

Figure 5.17: Execution time vs packet loss rate(%) between TCP and the library.

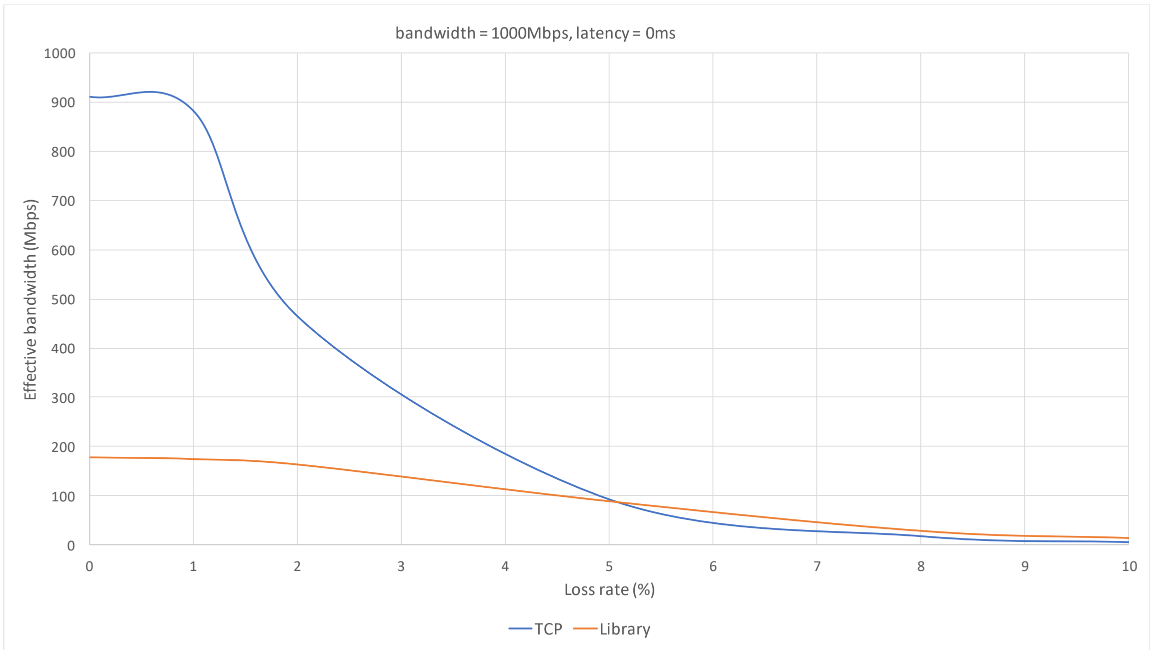

Figure 5.18: Effective bandwidth vs packet loss rate(%) comparison between TCP and the library.

5.6.4 Realistic conditions

We modeled four popular network technologies, using the three parameters we investigated above, with bandwidth getting split into upload speed and download speed because those are not equal in some of them, in `Mininet` and we ran the experiment for each one of them, resulting in Figures 5.19 and 5.20. The parameter combinations we used are shown in Table 5.3. So while using our library in fast networks is not practical or necessary, it provides a worthwhile performance improvement compared to plain TCP in networks with adverse conditions.

Name	Latency (ms)	Download speed (Mbps)	Upload speed (Mbps)	Loss rate (%)
Gbps Ethernet	0.1	1000	1000	0
WiFi	10	40	40	1
4G	25	20	6	0
5G [18]	13	300	64	0

Table 5.3: The sets of Mininet parameters we used to model popular network technologies.

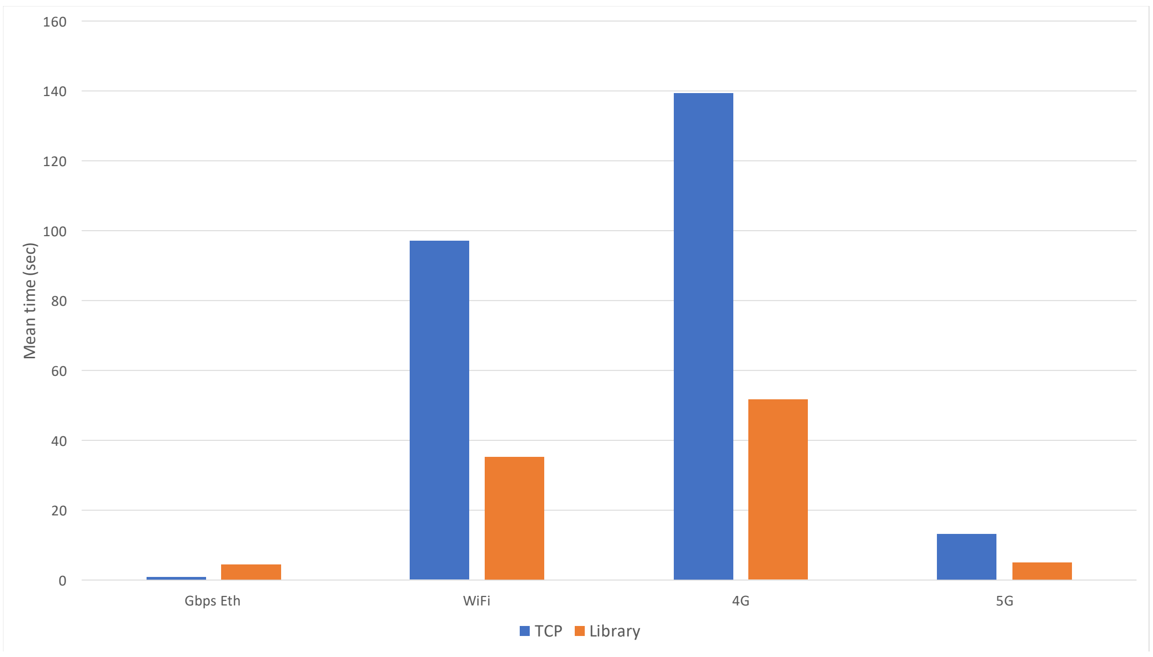

Figure 5.19: Execution time comparison for various network technologies.

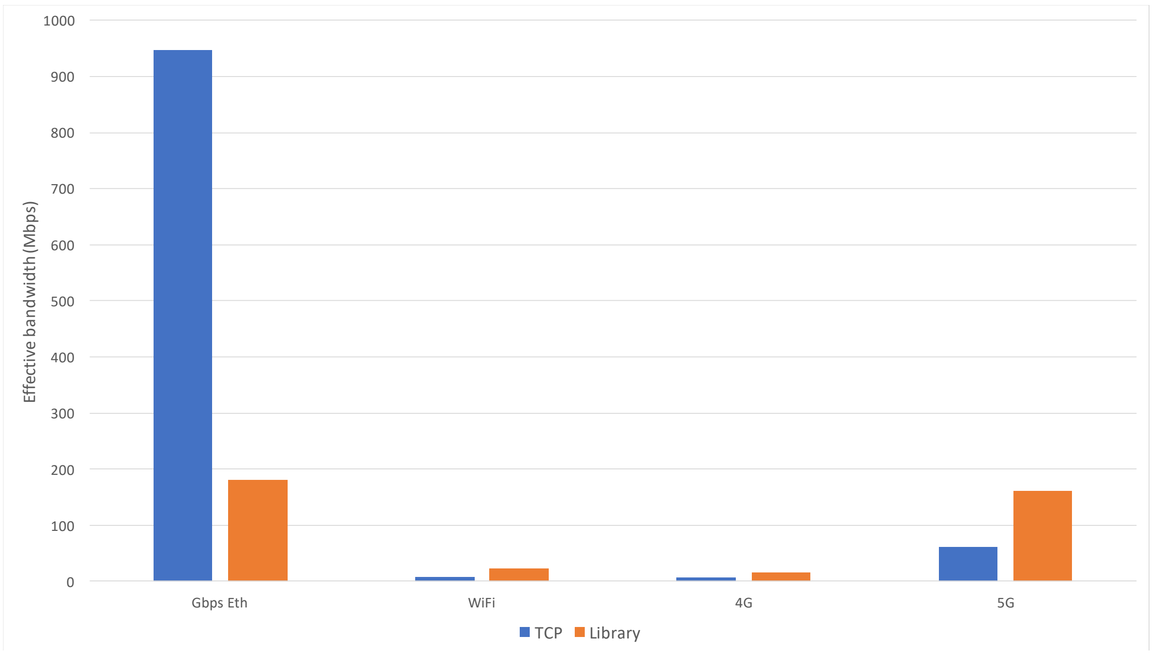

Figure 5.20: Effective bandwidth comparison for various network technologies.

Chapter 6

Related work

In this chapter, we discuss other attempts to use data compression in order to increase performance in low-bandwidth networks. Some of them run in the application layer just like ours, whereas others run in different layers of the network stack.

6.1 Application layer solutions

Jeannot [19] developed a user-level communication library using lossless data compression. This library is not transparent to applications (unlike ours), but it uses threading for concurrent compression and sending of data and can change the compression level based on how fast the network is, using the AdOC algorithm. Also, they implemented a compression decision mechanism, which decides whether a block of application data is to be compressed or not, based on the compression efficiency for the previous blocks.

Krintz, C. and Sucu, S. [20] designed and implemented ACE, a system that intercepts TCP socket calls and uses system and network predictions from the Network Weather Service to determine whether to compress or not, and which algorithm to use. ACE uses a 32KB block size because they concluded it exhibited the best trade-off across file types and metrics, compared to our `BUF_SIZE = 204800 B` which we selected based on compression ratio only.

Gutwin, C. at al. [21] designed a system for use with groupware, which compresses groupware messages using the redundancy between messages, by automatically detecting the message formats used by the groupware, and the redundancy within each message similarly to how we compress each buffer.

HTTP [22] includes the option to specify that the payload is compressed using compression algorithms such as gzip, in order to reduce the bandwidth needed to fetch the resources. FTP [23] features a compression mode, where run-length encoding is used to compress sequences consisting of repeated characters.

6.2 Solutions in other layers

Min Wang et al. [24] proposed TCPComp, a data compression scheme in the transport layer (whereas our implementation resides in the application layer). Their compression unit size is not fixed, compared to our implementation where it is `BUF_SIZE`, but computed according to a compression ratio estimate and the TCP maximum segment size (MSS).

Moo-Yeol Lee et al. [25] also suggested a kernel-level TCP data compression scheme while focusing on mobile devices communicating over WiFi. They implemented and evaluated both using MSS as the compression unit and using every application message as a separate compression unit, with the second approach performing better. In our implementation, we compress `BUF_SIZE B` of application data and pass it to TCP.

V. Jacobson [26] proposed a method for compressing TCP/IP headers, to improve performance when the bandwidth is low, by omitting header fields that are constant during a TCP session. A. Shacham et al. [27] proposed IPComp, a protocol that uses data compression on the IP datagram payload.

In the data link layer, CCP [28] is a method to negotiate data compression for communication over a PPP link. Cisco [29] internetworking devices use lossless data compression algorithms to reduce the bandwidth needed to transmit a frame, and speed-up communications in the same way as our implementation.

Chapter 7

Conclusion

In this Thesis, we focused on mitigating the performance degradation caused by low bandwidth links in the network edge by reducing the amount of transmitted data. To that end, we designed and implemented a library that performs lossless data compression over TCP. Afterwards, we validated the correctness of our library and evaluated its performance using a testing platform we developed over Mininet's network emulation.

By comparing the performance of our library with that of plain TCP in various network conditions, we conclude that our library is worth using over plain TCP when the bandwidth is low, confirming our initial hypothesis. We also observe that our library is preferable to plain TCP when the packet loss rate is high.

Finally, by measuring the library's CPU utilization in different network conditions, we observe that in low bandwidth or high packet loss rate networks, the CPU utilization decreases considerably. From this result, we conclude that while the CPU-bandwidth trade-off for compression exists, considering that our library is intended for low-bandwidth networks, the CPU load is low and compression can be a viable strategy. Further experiments need to be performed in order to determine whether compression could be used in the edge devices, which have weaker CPUs than the one we used.

Bibliography

[1] David Gibson and Ben Popper. Pandemic lockdowns accelerated cloud migration by three to four years. `https://stackoverflow.blog/2021/09/02/pandemic-lockdowns-accelerated-cloud-migration-by-three-to-four-years/`. Retrieved March 21, 2022.

[2] Gartner forecasts worldwide public cloud end-user spending to reach nearly $600 billion in 2023. `https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023`. Retrieved February 22, 2022.

[3] Peshraw Ahmed Abdalla and Asaf Varol. Advantages to disadvantages of cloud computing for small-sized business. In *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, pages 1–6, 2019.

[4] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[5] Rachel Azafrani, John Barett, and Hanane et al. Becha. The internet of things: Applications for business. Technical report, The Economist Intelligence Unit, 2020.

[6] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[7] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[8] Khalid Sayood. 1 - introduction. In Khalid Sayood, editor, *Introduction to Data Compression (Third Edition)*, The Morgan Kaufmann Series in Multimedia Information and Systems, pages 1–11. Morgan Kaufmann, Burlington, third edition edition, 2006.

[9] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[10] Khalid Sayood. 4 - arithmetic coding. In Khalid Sayood, editor, *Introduction to Data Compression (Third Edition)*, The Morgan Kaufmann Series in Multimedia Information and Systems, pages 81–115. Morgan Kaufmann, Burlington, third edition edition, 2006.

[11] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[12] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[13] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[14] P. Deutsch. Deflate compressed data format specification version 1.3. `https://www.rfc-editor.org/info/rfc1951`, 1996.

[15] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. page 19, 10 2010.

[16] Mininet python api reference manual. `http://mininet.org/api/`. Retrieved February 20, 2022.

[17] The canterbury corpus. `https://corpus.canterbury.ac.nz/index.html`. Retrieved February 22, 2022.

[18] Therdpong Daengsi, Pana Ungkap, and Pongpisit Wuttidittachotti. A study of 5g network performance: A pilot field trial at the main skytrain stations in bangkok. In *2021 International Conference on Artificial Intelligence and Computer Science Technology (ICAICST)*, pages 191–195, 2021.

[19] E. Jeannot. Improving middleware performance with adoc: an adaptive online compression library for data transfer. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10 pp.–, 2005.

[20] C. Krintz and S. Sucu. Adaptive on-the-fly compression. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):15–24, 2006.

[21] Carl Gutwin, Christopher Fedak, Mark Watson, Jeff Dyck, and Timothy Bell. Improving network efficiency in real-time groupware with general message compression. pages 119–128, 11 2006.

[22] J. Mogul et al. R. Fielding, J. Gettys. Hypertext transfer protocol – http/1.1. `https://www.ietf.org/rfc/rfc2616.txt`, 1999.

[23] J. Reynolds J. Postel. File transfer protocol (ftp). `https://www.ietf.org/rfc/rfc0959.txt`, 1985.

[24] Min Wang, Junfeng Wang, Xuan Mou, and Sunyoung Han. On-the-fly data compression for efficient tcp transmission. *KSII Transactions on Internet and Information Systems*, 7(3):471–489, March 2013.

[25] Moo-Yeol Lee, Hyun-Wook Jin, Ikhwan Kim, and Taehyoun Kim. Improving tcp goodput over wireless networks using kernel-level data compression. In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–6, 2009.

[26] V. Jacobson. Compressing tcp/ip headers for low-speed serial links. `https://www.rfc-editor.org/rfc/rfc1144`, 1990.

[27] R. Pereira M. Thomas A. Shacham, R. Monsour. Ip payload compression protocol (ipcomp). `https://www.rfc-editor.org/rfc/rfc3173`, 2001.

[28] D. Rand. The ppp compression control protocol (ccp). `https://datatracker.ietf.org/doc/html/rfc1962`, 1996.

[29] Understanding data compression. `https://www.cisco.com/c/en/us/support/docs/wan/data-compression/14156-compress-overview.html`. Retrieved March 20, 2022.

APPENDICES

A.1 Code snippets

A.1.1 Send implementation

In the following simplified version of my send function, we have abstracted the above functionality into a `compress_and_send(buf, len)` function, so as to highlight the buffering logic.

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags) {
    size_t rem_len = len;

    if (len + send_len[sockfd] < BUF_SIZE) {
        memcpy(send_buf[sockfd] + send_len[sockfd], buf, len);
        send_len[sockfd] += len;
        rem_len = 0;
    }
    else if (send_len[sockfd] > 0) {
        memcpy(send_buf[sockfd] + send_len[sockfd], buf, BUF_SIZE - send_len[sockfd])
        compress_and_send(send_buf[sockfd], BUF_SIZE);
        rem_len = rem_len - BUF_SIZE + send_len[sockfd];
        send_len[sockfd] = 0;
    }

    while (rem_len >= BUF_SIZE) {
        compress_and_send(buf + len - rem_len, BUF_SIZE);
        rem_len -= BUF_SIZE;
    }

    if (rem_len > 0) {
        memcpy(send_buf[sockfd], buf + len - rem_len, rem_len);
        send_len[sockfd] = rem_len;
    }

    return len;
}
```

A.1.2 Recv implementation

In the following simplified version of my recv function, we have abstracted the receiving from TCP and decompression functionality into a `receive_and_decompress(buf, &len)` function, so as to highlight the buffering logic. Also, we have omitted the error and EOF handling to simplify the presentation.

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags) {
    uLong decomp_len;
    size_t total_recv = 0;

    if (len < recv_len[sockfd] - recv_pos[sockfd]) {
        memcpy(buf, recv_buf[sockfd] + recv_pos[sockfd], len);
        recv_pos[sockfd] += len;
        total_recv = len;
    }
    else {
        if (recv_len[sockfd] - recv_pos[sockfd] > 0) {
            memcpy(buf, recv_buf[sockfd] + recv_pos[sockfd], recv_len[sockfd] - recv_pos[sockfd]);
            total_recv += recv_len[sockfd] - recv_pos[sockfd];
            recv_pos[sockfd] = recv_len[sockfd] = 0;
        }

        _flush(sockfd);
    }

    while (len - total_recv >= BUF_SIZE) {
        receive_and_decompress(buf + total_recv, &decomp_len);
        total_recv += decomp_len;
    }

    if (total_recv == len) {
        return total_recv;
    }

    receive_and_decompress(recv_buf[sockfd], &recv_len[sockfd]);

    if (len - total_recv < recv_len[sockfd]) {
        memcpy(buf + total_recv, recv_buf[sockfd], len - total_recv);
        recv_pos[sockfd] = len - total_recv;
        total_recv = len;
    }
    else {
        memcpy(buf + total_recv, recv_buf[sockfd], recv_len[sockfd]);
        total_recv += recv_len[sockfd];
        recv_len[sockfd] = 0;
    }
    return total_recv;
}
```