UNIVERSITY
OF OSLO

**Master's thesis**

# TCP PEP

TCP Performance Enhancing Proxy to Support
Non-interactive Applications

**Joe Bayer**

Informatics: Programming and System Architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences


Autumn 2023

**Joe Bayer**

# TCP PEP

TCP Performance Enhancing Proxy to
Support Non-interactive Applications

Supervisors:
Michael Welzl
Kristjon Ciko

# Contents

# List of Figures



## List of Listings

**Abstract**

...

# Chapter 1

# Intro

## 1.1 Motivation

# Chapter 2

# Background

In this chapter we will present some of the required background knowledge to understand the concepts presented in this paper. Focusing on topics that are outside the common understanding of network programming, especially details of certain congestion controllers and network protocols will be discussed. The rest of the thesis will assume the following topics are known to the reader.

## 2.1 TCP/IP

Perhaps the most well known internet transport protocol is the Transmission Control Protocol (TCP). It is known for providing reliable and in-order delivery of packets using acknowledgments and re-transmissions [6]. It was first introduced in 1974, but is still one of the most used internet protocols. However, as the demands of the internet have changed, TCP has not. Though TCP has been updated with minor extensions over the years, such as an increased initial window or new options, the core ideas have stayed the same [2].

Concepts as the end-to-end argument still play a vital role in how TCP is used in the modern internet. TCP is suffering under the illusion that all logic should be placed on the endpoints as the end to end argument denotes.[1] TCP often spans multiple different domains with varying topologies and demands, especially between wired and wireless domains. ...

- **Wireless Domain**: A wireless communication domain refers to the transmission of data over a wireless medium without the use of physical connections such as wires or cables between devices. This domain covers a variety of technologies, including 3G, 4G, and 5G for mobile

[1] Biased, TCP doenst feel

communication, Bluetooth and Wi-Fi for close-range communication, and satellite communication for worldwide communication.

- **Wired Domain**: Unlike a wireless domain, a wired domain provides a steady and reliable bandwidth with low error rates and high throughput. The use of Ethernet and Fiber are typical for wired networks, they enable the transmission of a large amount of data over long distances with low signal noise.

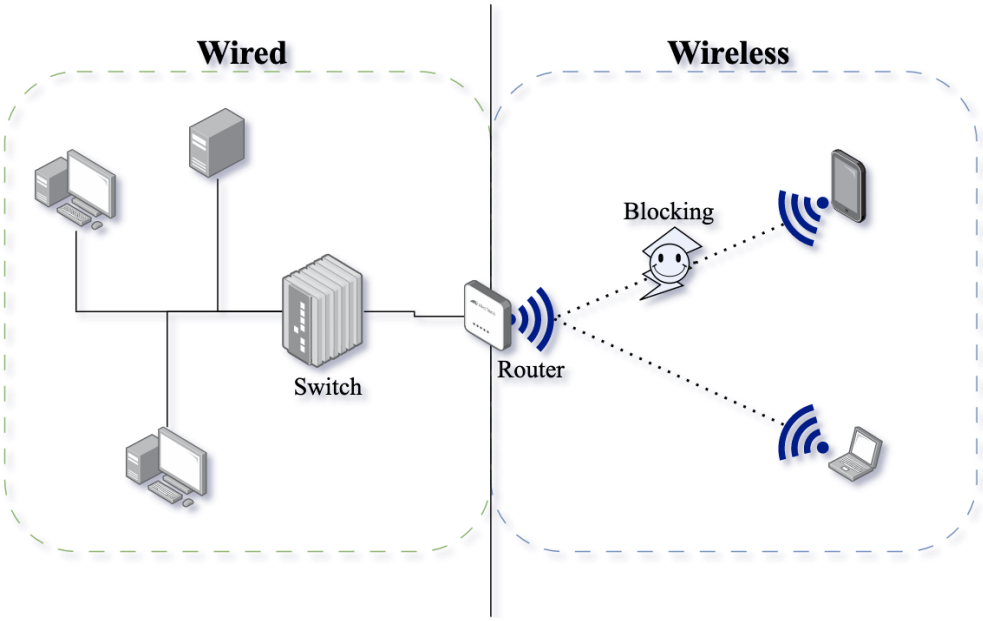


Figure 2.1: Example of network domains

Each domain has different requirements that a single TCP connection cannot satisfy. Fig. 2.1 shows the two domains and their characteristic differences. Usually, wireless domains experience a lot of changes in connectivity and bandwidths, while the wired domain is usually considered stable. This creates problems for TCP which normally spans multiple domains. Especially congestion control has problems adapting to highly fluctuating bandwidth across long distances and multiple domains.

### 2.1.1 Congestion Control

Congestion occurs in the internet when a network's resources, such as routers, are overloaded to the point that they diminish quality of the network [16]. Packet loss and delays are common issues associated to congestion in the network. To solve the problem of congestion, a distributed algorithm is used: Congestion Control. The main goal of congestion control is to maintain a

stable network, while still utilizing the available bandwidth shared among all flows. This is achieved by for example: additively increasing the sending rate, and multiplicatively reducing the sending rate when detecting congestion [15]. Congestion can be detected by monitoring packet loss, changes in delay, but also by explicit notifications.

Over time different variations of congestion controllers have emerged. Although their goal is the same: reduce congestion in the network, their approaches vary. Here are three examples:

- **TCP Reno**: Reno embodies the traditional approach to congestion control: slowly increasing the sending rate while the network is stable and drastically reducing it on packet loss. TCP Reno was designed for unstable and dynamic networks, where the rapid response rate is crucial to prevent network overloading. However, the slow start rate and aggressive reduction of the sending rate make it sub optimal for more stable networks[2], where packet loss is less frequent and predictable. Consequently, TCP Reno's reliance on packet loss may lead to unnecessary rate reductions and decreased network throughput.
- **Vegas**: Vegas is similar to TCP Reno in most aspects, the main difference is the use of delay to detect congestion instead of packet loss. This makes New Vegas able to react faster to congestion, however it also introduces some problems. If Vegas competes with TCP Reno flows, it will start reducing its sender rate before TCP Reno does, this leads to Vegas losing out on possible bandwidth.
- **Cubic**: Cubic improves on the idea of TCP Reno by using a cubic function to adjust its sending rate in order to achieve higher throughput in a fast manner. Cubic is very efficient in highspeed networks and known for handling large data transfer over long distances. However, Cubic is not as reliable and robust as more traditional congestion controllers like TCP Reno.[3]

In summary, the main differences between TCP Reno, Vegas and Cubic are their approach to congestion control, their performance in different types of networks, and their trade-off between efficiency and reliability.

### 2.1.2 3 Way handshake (0 RTT)

For TCP to establish a connection it uses a three-way handshake. Initially, it transmits a synchronization (SYN) packet to the desired endpoint. The endpoint responds with an acknowledgement and a synchronization packet

[2]This is not true, its bad with high BDP

[3]Find source, reason for this sentence is that the window changes more radically?

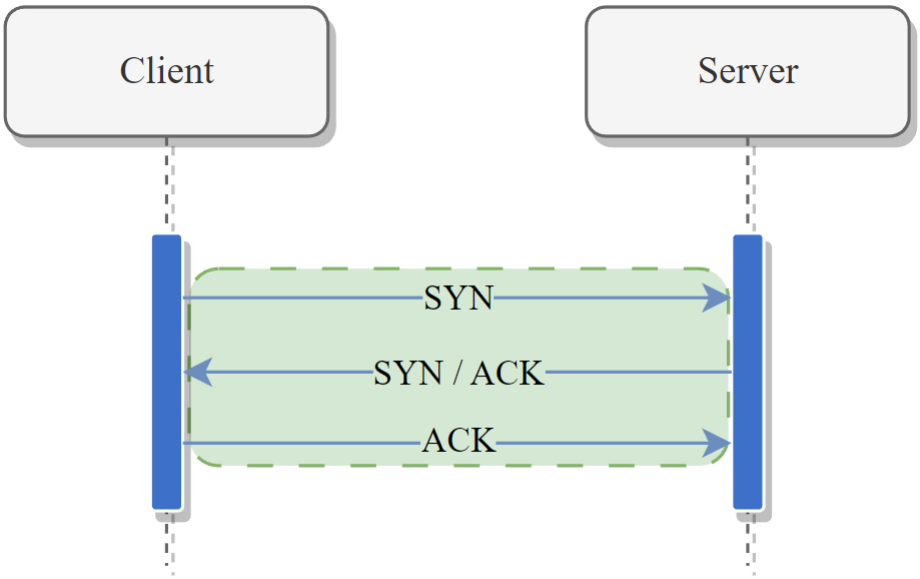


Figure 2.2: The TCP handshake procedure

of its own (SYN/ACK). Finally, the client responds with a acknowledgment (ACK). At this point both endpoints have confirmed that they are ready for further communication. For any connection to be established this handshake has to be done. For short flows that terminate in just a few round trips the initial TCP handshake can be a bottleneck, which is made worse if the connection is using a proxy[4] and has to exchange additional information.

### 2.1.3 TCP Options and Fast Open

A TCP connection can be configured with optional header extensions called TCP Options [3]. These options change the default behaviour of TCP or add new features. One such feature is TCP Fast Open, which allows data to be added to the initial synchronization packet. A typical use case could be adding a HTTP GET request, thereby saving an entire round trip. In general, flows that terminate in a few round trips greatly benefit from this feature because the bottleneck is within the initial TCP handshake. Therefore, by removing the extra round trip required to send the first data packet, a significant amount of time can be saved.

TCP Fast Open also has other benefits, such as establishing connections to proxies [2]. When one is establishing a connection through a proxy, one gets the added delay of a second round trip for sending the desired endpoint.

[4]Proxies have not been introduced, can I assume people know what a proxy is?

This can be avoided by using TCP Fast Open to send the desired endpoint in the first synchronization packet to the proxy. "SYN forwarding" enables the users to establish a proxy connection without any added delays, however it does depend on the user's application to use TCP Fast Open.

## 2.2 The future of wireless communication.

Wireless communication has seen a lot of improvements such as highly increased bandwidth achieved through advanced technologies like 5G. Millimetre frequency bands have opened up new possibilities for wireless communication. These higher frequency bands offer greater capacity and can accommodate more devices, however high frequencies come with a set of new challenges such as highly fluctuating bandwidths. This fluctuation can be influenced by various factors such as signal interference, obstacles in the signal path, and environmental conditions.

### 2.2.1 5G Millimetre Wave

The emergence of 5G Millimeter wave communications has opened the doors for low latency networks with multiple gigabits bandwidth. This is achieved by using higher millimetre wave (mmWave) frequencies in the range of 30GHz to 300GHz, which has a lot of benefits [1]. A wider spectrum of frequencies to choose from and higher data transfer rates are just some of the many benefits mmWave provides. But alongside the benefits, mmWave has also introduced a lot of new challenges.

A big problem with millimetre wave communication is signal path blocking, also called "Line of sight blocking" [13]. It's caused by the use of Beam-forming to increase the bandwidth and range of milimeter wave signals. Beam-forming focuses the signal in a certain direction, making any blocking of the signal path devastating for the bandwidth. Even the human body can create enough blockage to drastically reduce the bandwidth. This causes huge fluctuations in the bandwidth whenever the signal is blocked.

Fluctuating bandwidths lead to unstable TCP connections with a worst case of losing packets. Current TCP congestion controllers such as CUBIC, Reno or Vegas struggle when reacting to sudden changes. They are simply not able to utilize the high bandwidth when it is available. Increasing the aggressiveness of a congestion controller is not an option either, as it would disrupt the internet. A possible solution could be to buffer packets at the 5G base stations, having the data ready for when the bandwidth is high. This however creates a new problem: "bufferbloat".

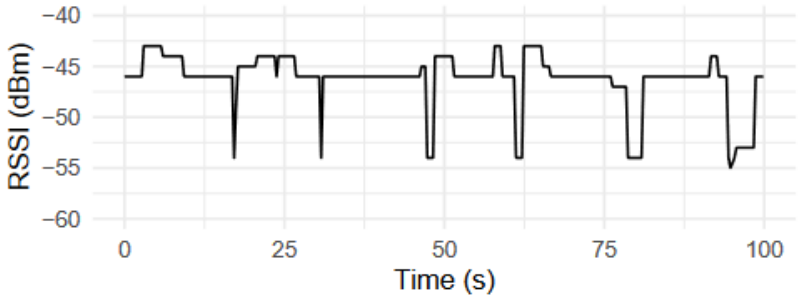


Figure 2.3: 5G bandwidth fluctuations from humans

### 2.2.2 Buffering

#### Bufferbloat

The bufferbloat problem occurs when the systems between the endpoints buffer so many packets that the latency drastically increases and the reliability of the network as a whole goes down. The increased latency is detrimental for interactive (latency sensitive) applications. Generally it's preferred to drop packets and keep buffers small to avoid buffering time sensitive packets such as synchronization packets. Although this works in most cases, it's far from a optimal solution.

The increased bandwidth and low latency promises of new technology such as 5G has put a lot of pressure on the efficient forwarding of packets. Small buffers are therefore the standard, but at the same time, fluctuating bandwidth has shown the potential need to buffer packets for non-interactive traffic. Most focus has been on accommodating latency sensitive applications like virtual reality or remote surgery to name a few.

This thesis will explore non-interactive applications where latency is not that critical and more buffering is acceptable and most likely desirable. By splitting traffic into interactive and non-interactive we can improve the performance of both. By having very small buffers for interactive applications we avoid bufferbloat problems, while utilizing the benefits of big buffers for non-interactive applications.

#### Active Queue Management

#### Packet Scheduling

A method of reducing the effects off bufferbloat is packet scheduling. A system should not send more packets than the weakest link can handle; this idea is built into TCP in the form of congestion control. However, when buffers

grow to the point of causing bufferbloat, TCP's congestion control algorithms are unable to confidently determine a sending rate. Packet scheduling can solve this problem as it controls the size of the buffers. It makes sure queues can grow when needed, but keeps the overall state of the buffers low. Packet scheduling has a lot more to offer than simple queue management, this will be explored later.

Proposed algorithms:

- **FQ_CoDel**: The Flow Queue Controlled Delay algorithm, FQ_CoDel for short, was developed to deal with the bufferbloat problem. Its main goal is to reduce the impact of head-of-line blocking and give a fair share of bandwidth by mixing packets from multiple flows [9]. Internally FQ_CoDel uses a FIFO queue, classifying packets into different flows to provide a fair share of bandwidth.

- **HTB**: Hierarchical Token Bucket is a queuing discipline based on assigning different classes a certain amount of bandwidth and sending rate. Because of its extensive bandwidth and delay management it's a good option for testing, especially in a virtual environment.

### 2.2.3 Non-Interactive Applications

Non-Interactive applications such as web traffic, file transfers and video streaming can benefit from larger buffering, especially with fluctuating bandwidths. This is because, if we are able to buffer the packets closer to their final destination, we have them ready to be sent when the bandwidth changes. As this thesis will show, by buffering them we can decrease delay times and achieve faster total completion times for non-interactive traffic.(need citation or prove it myself?). At the same time, with the solution presented in this thesis, interactive applications will not suffer under large queue delays that occur under normal buffering.

## 2.3 Proxy

Proxy servers play a big role in the modern internet, delivering benefits such as anonymity and increased performance [14]. A common use case for a proxy is caching by keeping a copy of popular resources such as a websites. This reduces the latency of accessing the resource as long as the proxy is closer to the user than the original copy. Locality plays an important role in the total latency as any transmission will always be limited by the speed of light.

A proxy can also be used for privacy similar to a Virtual Private Network (VPN). By redirecting network traffic through a proxy, the origin of

the traffic appears to be the proxy server rather than the actual end-user. Hypertext Transfer Protocol (HTTP), a popular internet protocol used for accessing websites, has this functionality built in using HTTP tunnels and a special CONNECT method in its header:

```
CONNECT mn.uio.no/:22 HTTP/1.1
Proxy-Authorization: Basic encoded-credentials
```

### 2.3.1 PEP

[5] A performance enhancing proxy (PEP) is a proxy designed to increase the performance of applications using it, typically by influencing the behavior of TCP. The idea behind the PEP is putting more logic, such as connection management, buffering, caching inside the network. As the name suggests, a PEP is designed to enhance the performance, but can also introduce new features to a network. An example of a new feature is the multipath support the TCP Transport converter gives [2].

### 2.3.2 PEP for wireless communication

Performance enhancing proxies are already deployed and in use for a lot of wireless communication, especially satellites and radio access networks [12]. They have an inherent performance increase just by splitting the connection between the wireless and wired domains. These PEPs are therefore often installed at the base stations. However they are unable to distinguish between interactive or non-interactive traffic, meaning their buffers need to be small to avoid bufferbloat and hence why still suffer from fluctuating bandwidth problems.[6] Alternatively, a PEP with a large buffer can compensate for fluctuating bandwidth, but will introduce delay for interactive traffic.

### 2.3.3 Transparent vs Non-Transparent

A big discussion regarding PEPs has been if they should be transparent or non-transparent. Transparent PEPs are not visible to the applications that use it. They silently split the connections and spoof the IP-address of both the client and server [4]. This is prone to cause unintended side effects, such as certain TCP options not being forwarded, and security concerns. Non-Transparent PEPs on the other hand are explicitly chosen by either the client or the server, and the sender is aware of the proxy splitting the original connection. This approach can be seen as more appropriate for the internets architecture and could potentially remove some of the stigma associated with

[5] Include that they need to be on path to be "fast"

[6] Confusing sentence?

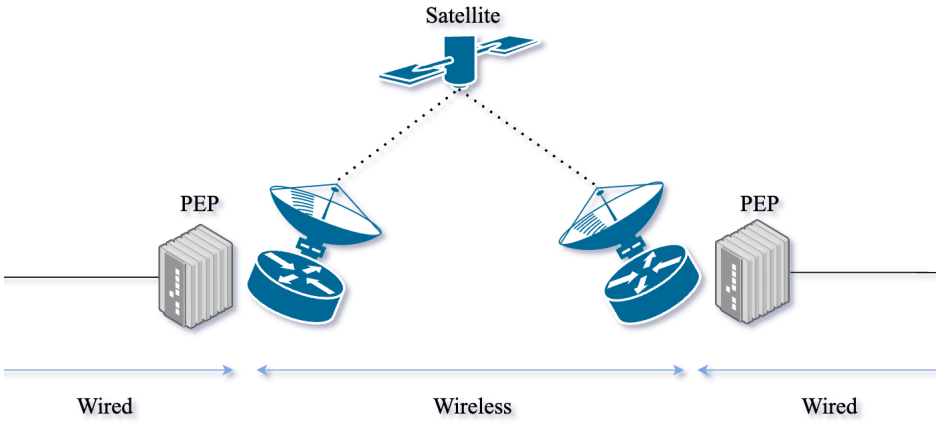


Figure 2.4: PEP installed to support Wireless traffic over satellite.

PEPs. This however requires modifications at the sender side to utilize the PEP.

## 2.4 Linux

Linux is the most famous open source kernel freely available for anyone to use and modify. Because of the open source nature of Linux, there have been many operating system implementations based on the Linux kernel. Ubuntu, Fedora or Manjaro are just some of the most famous Linux based operating systems out there. For developers, Linux is the perfect platform to experiment and test their new innovations. One is able to modify and recompile the kernel itself on the fly, and then test the solution on a live operating system. Linux supports most standards and is used by most major corporations such as Facebook, Amazon, Netflix and Google.

### 2.4.1 Kernel Modules

A concept that makes Linux truly extensible are Loadable Kernel Modules (LKM). Kernel modules are programs that can be loaded at runtime into the kernel and run with kernel privileges. Running with kernel privileges has a lot of benefits such as having access to internal structures and kernel symbols. Most drivers in the Linux kernel are written as kernel modules as they need access to the system internals.

Congestion controllers and packet schedulers are also usually implemented as kernel modules. That is because Linux exposes a struct with function pointers that can be overwritten by a module, making the kernel call the new functions instead. Because kernel modules run as part of the kernel

they do not need to use a system call to do basic I/O as using sockets. Removing the overhead of system calls makes the kernel modules run much faster than default user space programs.

However, using Linux kernel modules has the drawback that the program is bound to Linux. The modules will only work in the context of the Linux kernel as they depend on the internal functions, and that they are part of the kernel. Most other operating systems like MacOS will not allow user defined modules to run with kernel privileges. Additionally, any bugs or error in the kernel module with make the entire kernel fail ("panic"), which usually requires a complete system restart to fix.

### 2.4.2 C Programming Language

C has been the optimal language for high performance systems since its creation in 1972. [10] It was originally created for UNIX when it needed a higher level language, and now is the main programming language behind most operating systems such as Linux, Mac and Windows. Being very close to its predecessor, assembly, and compiled to a binary, makes it one of the fastest languages we have to date. Heap memory management is explicitly done by the programmer with no support for garbage collection. The unsafe memory management is one of the main challenges when programming in C, which however can be a benefit because a runtime garbage collection usually results in performance loss.

Listing 2.4.1: Default C program.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   printf("Hello World!");
6   return 0;
7 }
```

## 2.5 Related Work

PEPDNA? 0RTT transport converter?

# Chapter 3

# Design

In this thesis we design a non-transparent connection splitting PEP at the boundary of an unstable network with fluctuating bandwidth, typically at 5G base-stations. By splitting the connection at this point we can buffer packets, usually closer to the user, and have them ready for high bandwidth phases. Where the overarching goal is to improve the completion times of non-interactive traffic while avoiding to disturb the interactive flows that are passing through. A good design for the PEP is crucial as it needs to be robust, fast and reliable.

Figure 3.1 is a simplification of the PEP within a network, highlighting its interaction with other components, such as the client and endpoint.

## 3.1 Justification for designing a PEP

The ossification of networks and particularly TCP, have been long-standing issues [8]. Over the years, the internet has evolved, but the core protocols, like TCP, have remained relatively unchanged. This leads to challenges when attempting to introduce extensions or modifications. Altering such a fun-

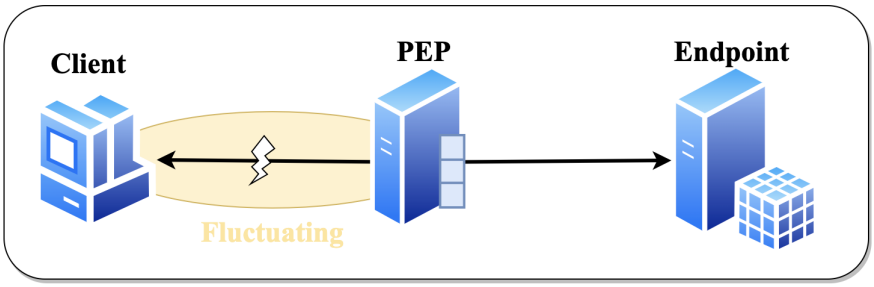


Figure 3.1: PEP within a network

damental protocol could disrupt countless systems and applications. Which leaves us to explore new ideas using middle-boxes. A PEP is such a middle-box, as it can enhance the performance without introducing changes to TCP itself.

Using a PEP in combination with 5G has additional benefits. Default TCP with 5G needs to cover both the stable network and the fluctuating wireless domain split by base station[1]. However, with a PEP we are able to split the domains and perform optimizations, such as congestion control and buffering, tailored to each specific domain. Achieving the same optimization by modifying TCP would need a change to the tight integration of end to end congestion control.
[2]

## 3.2 Technical Design choices

In this section, we discuss the core design decisions tailored to optimize the PEPs performance. The speed at which a PEP can both process and forward packets is crucial. Especially when wanting to utilize the rapidly fluctuating bandwidth of 5G, we need to react as fast as possible. The design choice presented in this section focus on the performance of the PEP. Because of these high performance requirements the PEP is written in C as a Linux kernel module.

### 3.2.1 Programming Language

The programming language chosen for a PEP has a direct impact on its efficiency. Interpreted languages like Python might not offer the speed necessary for high-performance tasks. Even Java, while running within the JVM with JIT and garbage collection, can potentially introduce delays. The languages best suited for high performance are C, C++ and Rust. Both C and C++ are very similar and well suited for high performance systems. The reason for choosing C is its bare metal approach and integration into the Linux kernel with kernel modules.

### 3.2.2 Kernel Module Vs. Userspace Application

When sending data over the network, all calls to receive and send data has to go through the kernels internal network stack. These calls are usually in the form of system calls when running as a userspace application. Making system calls (syscalls[3]) can introduce a lot performance overhead in the form

[1] Is it a basestation?
[2] PEPs with 5G, different "domains" works better than normal PEPs
[3] Section about system calls?

of a context switch to kernel space and copying data back and fourth from user-space to kernel-space. A kernel module will be able to directly access the kernel functions, eliminating the overhead of system calls and the restricted access of userspace applications to the network stack.

Opting for a user-space application has the advantage of being cross-platform, meaning it can run on multiple operating systems without major modifications. In contrast, kernel modules are tightly bound to the Linux environment, which might limit its usability to Linux hosts. This however is not a big problem as most servers are Linux hosts. [CITE]

### 3.2.3 Connection Splitting using Sockets

A connection splitting proxy will additionally split the connection into different domains. As discussed in Chapter 2, the internet consist of different domains with their own characteristics. Being able to split the connection into their different domains, enables the PEP adapt to the domains and to select an appropriate congestion controllers based on the technology and topology of each domain.

The PEP uses sockets to establish and split the connection between the client and the endpoint. Sockets provide a standardized way for programs to send and receive data over a network. They act as communication points, allowing for data exchange between them. For the PEPs purpose, sockets are ideal because they are configurable and come with dedicated buffers. This makes it easier to split and manage connections efficiently. Additionally, sockets are widely supported, ensuring compatibility and ease of integration with various network configurations and protocols.

### 3.2.4 PEP Selection

The connection process to the PEP, and later on the endpoint, is established by informing the PEP of the desired endpoint of the client. This can be achieved by a variety of ways where the goal is to attach additional information to the default connection process of TCP. Preferably, we do not want the overhead of needing an entire additional round trip just to pass this information. Figure 3.2 visualizes the RTT overhead of sending this information (in red) to the PEP before application data can be sent.

Additionally, we do not want to change the normal socket based scheme of creating a connection either, as old applications would need to rewrite a lot of their code to adapt a new scheme. By providing a library for the client we can easy the challenges of integrating the PEP into existing environments. The library will provide a single function which mimics the default connection

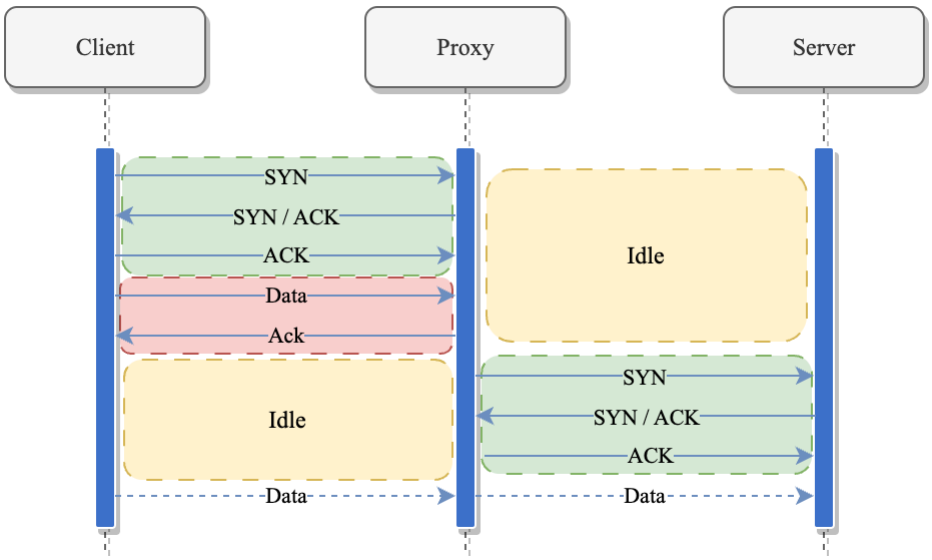


Figure 3.2: The TCP handshake procedure across PEP

scheme. Implementation details are discussed in Chapter 4.

### 3.2.5 Connection Establishment

The PEP will attach data to the initial TCP handshake, this way we can inform the PEP of the endpoint without needing to send it with an additional round trip time. Because of the ossified nature of TCP [8], changing the protocol itself is not an option. The realistic approach is reusing existing TCP functionality to append the desired data. Only a small amount of data, less than the usual Maximal Transmission Unit (MTU), is needed to inform the PEP of the endpoint.

**TCP Options** Since TCP Options can be attached to a TCP connection, a possibility would be to add a new TCP Option which would specify the endpoint. This TCP Option would need to be added by a kernel module, as it is not possible to to add custom TCP Options from user-space. This leads to another problem, mainly how to specify from user-space that we wish to use the PEP.

A possible option would be a socket option, using setsockopt. This however requires changes to the kernel, which raises the bar for adaptability. Another choice would be always attaching the TCP Options on connection addressed to a certain port such as 80/443. This however takes away the

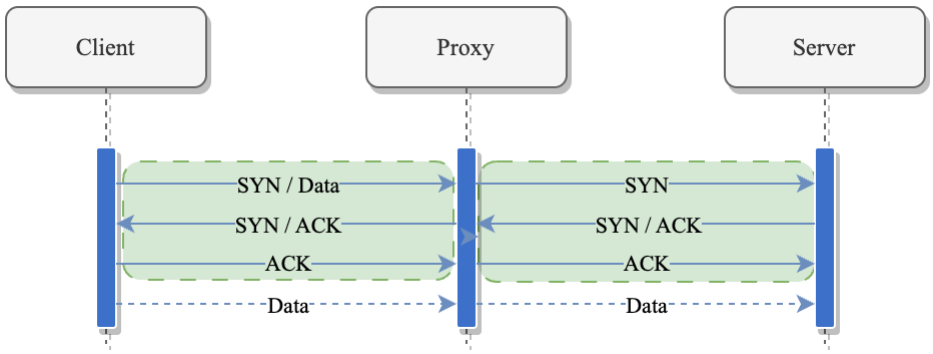


Figure 3.3: Optimal handshake across PEP (0 RTT)

choice from the application, and makes it system wide instead.

Finally, another significant problem is that unknown TCP Options are often seen as a threat. Firewalls may drop the packets, or the options might be stripped by intermediate nodes. [11] This creates a challenge for the implementation and usability of the PEP. If the packets may be dropped because of our custom TCP options, then the PEP will only work in certain networks and scenarios. Although we only design a proof of concept, this is a trade off that is unlikely to pay off in the end. [8] [4]

**TCP Fast Open** Another possibility is using the existing TCP Fast Open option which can attach data to the initial TCP handshake. As discussed in the Chapter 2, using TCP Fast Open can reduce the amount of RTTs needed to establish a connection with both the PEP and endpoint. This requires the socket to be configured and enabled system-wide on the server machine.

**Optimal Choice** The most sustainable choice is TCP Fast Open, as adding new TCP Options is simply too unstable. Also, the goal of our PEP is not to change or extend TCP itself. Using TCP Fast Open also has the advantage of being able to add meta data in the size of an MTU, which adds to the adaptability of the PEP to be extended in the future.[5]

Figure 3.3 shows the concept of the added data to the TCP handshake, reducing the idle time visualized in Figure 3.2.

[4] Directly quote the middlebox interaction paper section 4.4?
[5] Rephrase last sentence.

## 3.3 PEP Transparency

This thesis will utilize a non-transparent PEP because of the benefits of future adaptability and the client being aware of the PEP itself.

### 3.3.1 Transparent

Transparent PEPs provide a deployment advantage since they can be integrated without altering the client or server. This means even older applications can utilize the PEP. Moreover, the PEP will inherently be on the same path as the original connection, which is a prerequisite for a PEP. But, if unknown middle boxes interfere with a connection, issues like lost TCP Options might occur. Additionally, if applications are not aware of the PEP, they can not adjust their behavior, restricting their future adaptability.

### 3.3.2 Non-Transparent

Since applications are fully aware of the PEP when it is non-transparent, we can adjust the PEPs functionality based on the application using it. Applications can also adapt their behavior, optimizing their operations based on the presence of the PEP, leading to additional performance improvements. Although side effect may still occur, applications are aware of them and can actively mitigate them. Deployment becomes more problematic as for the PEP to be efficient it needs to be on the shortest path which is no longer guaranteed.

## 3.4 AQM & Scheduling Algorithms

The ability to configure and utilize different scheduling algorithms is important for the success of the PEP. This is another choice which will bind us to Linux, as achieving the same control and configuration on other operating system, such as windows or MacOS, will be extremely difficult or even impossible. The impact will be shown in the Evaluation chapter.

## 3.5 Buffering

Normally the context of buffers in the network, revolves around NIC buffers or routing buffers. Those buffers affect all traffic passing through and will cause Bufferbloat if the buffers are too large. That's why, normally, the goal is to keep these buffers low to avoid disturbing interactive traffic. Since the PEP is connection splitting we can utilize a different buffer, the socket buffers.

### 3.5.1 Socket buffers

A socket buffer is a buffer specific for a particular socket. There are usually two buffers, one for read on the socket and one for write. This offers a great opportunity for buffering only a certain type of traffic, as a socket buffer does not affect traffic which flows through the host machine. A connection splitting proxy inherently has two sockets, one for the client and one for the endpoint. Which means we get a total of 4 buffers per 'flow', each individual read / write buffer can configured for the best performance.

## 3.6 Security

## 3.7 Summary

Table of design decisions based on different PEP implementations compared to ours. 0RTT, Transparent, TLVs, Special ACKS, connection splitting.

| PEP List | | | | |
|---|---|---|---|---|
| Implementation | 0RTT | Connection Splitting | Special ACKs | Transparent |
| milliProxy | AF | AFG | 004 | x |
| PEPDNA | AX | ALA | 248 | x |
| SnoopTCP | AL | ALB | 008 | x |
| Our PEP | DZ | DZA | 012 | x |
| Transport Converter | AS | ASM | 016 | x |
| ... | AD | AND | 020 | x |
| ... | AO | AGO | 024 | x |

# Chapter 4

# Implementation

This chapter will explore implementation of the TCP PEP, following up on the design choices mentioned in the previous chapter. The development of the PEP will give a deeper understanding of the underlying mechanisms and how they aim to better utilize the 5G bandwidth. All aspects from kernel modules, PEP architectures and additional libraries will be covered.

## 4.1 Kernel Module

As mentioned in the design chapter, the PEP will be written as a kernel module instead of a normal user-space program. Running and creating a kernel module requires more initial preparation than the a normal application. Firstly, the biggest change is that our PEP will now be a run as a module inside the Linux kernel instead of as a application in its own virtual environment. Injecting a module into the Linux kernel is very different from simply running a binary.

A Linux kernel module is loaded and unloaded with the help of two functions that need to be defined:

Listing 4.1.1: The basic kernel module setup code.

```
1  /* Needed by all kernel modules */
2  #include <linux/module.h>
3  #include <linux/kernel.h>
4  #include <linux/init.h>
5
6  /* entry function */
7  static int __init onload(void) {
8      return 0;
9  }
10
11 /* exit function */
12 static void __exit onunload(void) {
13
14 }
15
16 module_init(onload);
17 module_exit(onunload);
```

Listing 4.1.1 shows a basic kernel module skeleton, defining and exporting the functions onload() and onunload(). The name of the functions bear no meaning, the important parts are the macros `__init`, `module_init` and the corresponding exit macros[5]. When a kernel module is loaded the function declared with `__init` is called. Normally a application would terminate when it returns from its `main` function. Kernel modules however remain "loaded" when returning from the initialization function. This brings us to a new paradigm when programming, instead of having a running program, we install hooks and callbacks which change the default behavior of the kernel. A callback architecture can be less resource intensive as we do not need threads polling for data.

A kernel module is loaded by first compiling it into a .ko file and then loading with the `insmod` shell command:

```
$ insmod <module_name>.ko
$ rmmod <module_name>.ko
```

The `rmmod` is used to unload a kernel.

### 4.1.1 Kernel Hooks

The Linux kernel exposes many function tables and callbacks which designate what functions to call at certain events. Most drivers and congestion controllers are implemented in this manner. A predefined struct is allocated and populated with custom functions, and installed with an existing kernel function.

Listing 4.1.2: Example of a TCP congestion controller module

```
1
2 static void my_init(struct sock* sk);
3 static u32 my_ssthresh(struct sock* sk);
4 ...
5
6 static struct tcp_congestion_ops mycc __read_mostly = {
7   .init           = my_init,
8   .ssthresh       = my_ssthresh,
9   .cong_avoid     = ...,
10  .set_state      = ...,
11  .undo_cwnd      = ...,
12  .pkts_acked     = ...,
13  .owner          = THIS_MODULE,
14  .name           = "tuner",
15 };
16
17 /* entry function */
18 static int __init onload(void) {
19     return tcp_register_congestion_control(&mycc);
20 }
21
22 /* exit function */
23 static void __exit onunload(void) {
24     tcp_unregister_congestion_control(&mycc);
25 }
26
27 module_init(onload);
28 module_exit(onunload);
```

In the Listing 4.1.2 we demonstrate how a TCP congestion controller is implemented, in the context of a kernel module. This allows run-time modification of kernel behaviour, and is a programming paradigm[1] which will be useful to use when implementing the PEP.

### 4.1.2 Linux Version and Distribution

### 4.1.3 Implications

```
1 int pep_tcp_receive(struct socket *sock, u8* buffer, u32 size)
2 {
3   struct msghdr msg = {
4     .msg_flags = MSG_DONTWAIT,
5   };
6
7   struct kvec vec;
8   int rc = 0;
9
```

[1] Is it a paradigm?

```
10    vec.iov_base = buffer;
11    vec.iov_len  = size;
12
13    printk(KERN_INFO "[PEP] kernel_recvmsg: calling recvmsg \n");
14  pep_tcp_receive_read_again:
15    rc = kernel_recvmsg(sock, &msg, &vec, 1, vec.iov_len,
        MSG_DONTWAIT);
16    if (rc > 0)
17    {
18      tlv_print(buffer);
19      printk(KERN_INFO "[PEP] kernel_recvmsg: recvmsg returned %d
        \n", rc);
20      return rc;
21    }
22
23    if(rc == -EAGAIN || rc == -ERESTARTSYS)
24    {
25      goto pep_tcp_receive_read_again;
26    }
27
28    printk(KERN_INFO "[PEP] kernel_recvmsg: recvmsg returned %d\n
        ", rc);
29    return rc;
30  }
```

Listing 4.1: kernel_recvmsg wrapper for receiving for TCP msgs

## 4.2 TLV Library

Regarding the endpoint addressing and selection by the client, a custom shared library is a good choice. [2] The goal is to keep the client code as simple and close to its original state as possible, but still be able to communicate the endpoint, potential options and meta-data to the PEP. Additionally, we want to send this information by using TCP Fast Open which has a similar connection routine as default sockets do.

Normally a socket would first create, connect and then send data using the `send` system call. The creation of a socket is the same for both default and TCP Fast Open. Which leaves us with connection as the main problem, the way we connect to a host using TCP Fast Open is by skipping `connect` and instantly jump to `sendto`. As we can see from the socket system calls, both `connect` and `sendto` take a `struct sockaddr` in as a parameter. This defines the endpoint to which you want to connect. `sendto` has a flag option which allows the configuration of how messages are sent, and if we supply the `MSG_FASTOPEN` flag, `sendto` will automatically connect and deliver the given message within the handshake. Subsequent uses of `send` will function

[2] Rewrite this.

as if the `connect` function was used.

**Important socket system calls**

```
int connect(
    int sockfd,
    const struct sockaddr *addr,
    socklen_t addrlen
);

ssize_t send(
    int sockfd,
    const void *buf,
    size_t len,
    int flags
);
ssize_t sendto(
    int sockfd,
    const void *buf,
    size_t len,
    int flags,
    const struct sockaddr *dest_addr,
    socklen_t addrlen
);
```

### 4.2.1 Custom connect function

The library will replace the original connect with an custom implementation `pep_connect`. The original `connect` and `sendto` have a lot of parameters in common, specifically the `const struct sockaddr` which is used to identify an endpoint. In the context of the PEP, this would identify the final endpoint to which the client wants to connect. The goal of the PEP connect function is to replace the `sockaddr` given by the client with one that identifies the PEP, but still forward the original `sockaddr` to the PEP to esatblish the proxy connection.

The signature of the our custom implementation mimics the original connect. The main difference is the addition of a flags parameters for easier customization. Inside our custom function we allocate space for a new `struct sockaddr` which we will fill with the IP address and port of the PEP, while at the same time we create a message with original IP address and port of the endpoint. Finally we call `sendto` with our message and the new `struct sockaddr`, using `MSG_FASTOPEN` to both connect and deliver the message.

**The custom connect function signature**

```
int pep_connect(
    int sockfd,
    const struct sockaddr* addr,
    socklen_t len,
    int flags
);
```

### 4.2.2 TLV

A good choice for sending options and meta-data is in the form of TLVs, formally known as Type–length–value options. The idea is that all options can be defined by a Type, Length and Value. The type defines the type of an option, what types exists and what they mean are up to the users of the library to decide. Common types are "Version", "Error", etc, adding new types is very easy and requires little modification. The TLV is practically implemented as its own message, but can also be appended at the start of a transmission like a header [2][3].

**TLV structures**

```
struct __tlv_header {
    unsigned char version;
    unsigned char len;
    unsigned short magic;
};

struct tlv {
    unsigned char type;
    unsigned char length;
    unsigned short value;
    unsigned int optional;
};
```

**TLV Implementation** In our design, TLVs are structured as a continuous buffer, consisting of a TLV header followed by subsequent options. This header provides details on the version, the number of options, and a unique magic number for validation. The `type` spans 1 byte, which means we limit ourselves to 255 possible types. The current implementation only uses 6, which means we have enough space for future extensions. The `length` variable is mainly used to indicate an optional data segment called `optional`. The size of the default value is 2 bytes with an additional option of size 4 bytes. All together the struct uses 8 bytes or 64 bits, which mostly comes from the fact we need atleast 6 bytes for the IP address and port alone.

[3]Reference this more directly as we adapt the idea of TLVs from this paper.

**Problems using TFO** When using TCP Fast Open ... instant return with no confirmation of a successful connection. [4]

### 4.2.3 TLV Options

The options of the TLVs define the functionalities a PEP can provide, for our PEP we only need 6 options. The basic `info` and `error` options are included alongside some information about the TCP connection such as extended headers. The most important type is `connect`, it specifies a port and IP address and is used to communicate the endpoint to the PEP. A TLV message can include as many options as a client wants.

```
enum __tlv_types {
    TLV_INFO = 0x1,       // Info TLV
    TLV_CONNECT = 0xA,    // Connect TLV
    TLV_EXT_TCP = 0x14,   // Extended TCP header
    TLV_SUPP_EXT = 0x15,  // Supported TCP extension
    TLV_COOKIE = 0x16,   // Cookie TLV
    TLV_ERROR = 0x1E    // Error TLV
};
```

### 4.2.4 Shared Library

The PEP TLVs are implemented as a shared library which is both used by the PEP itself and applications. The applications will use the library to create TLVs for connecting to the PEP, while the PEP uses the library to validate and read the TLV options. A shared library can be created by passing the `-shared` flag to the linker.
[5]

```
$ gcc file.c <path>/<lib>.so -o file.o
```

## 4.3 PEP - Internals

The internals of the PEP will consists of many important components. Importantly we have the sockets, socket pairs (tunnels) and deferred works. The PEP itself will need to keep track of its state and the state of all its tunnels, additionally we need to keep track of all running tasks. Since out code will be a kernel module and running 'inside' the kernel, we will have access to a lot of existing infrastructure which normally is only accessible by the kernel.

[4] Lots to add here...

[5] More about how to create shared libraries, include files etc. How to bind lib to application.

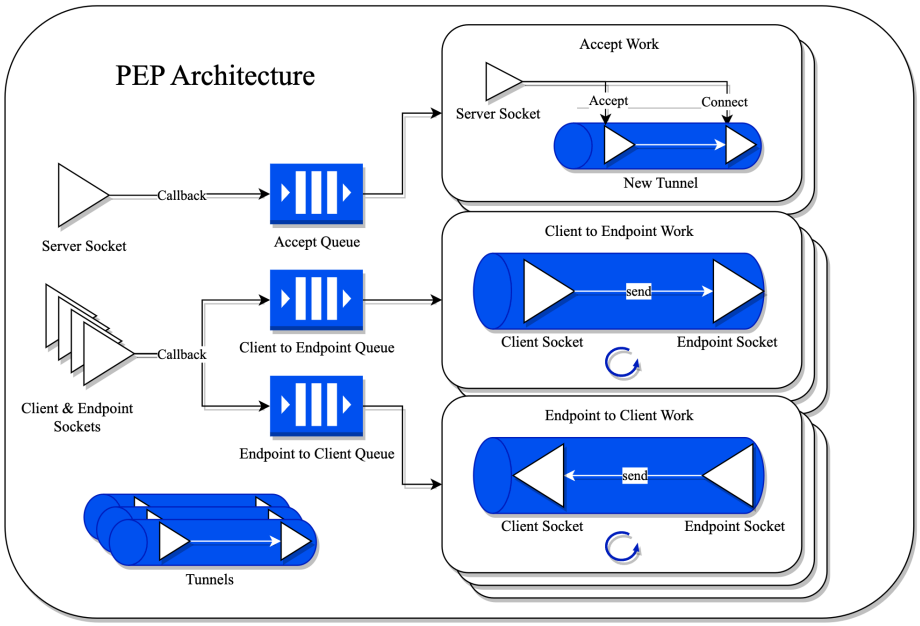


Figure 4.1: The architecture of the PEP

### 4.3.1 Architecture

The PEPs architecture is mainly based on a main server and multiple socket tunnels. The main server acts as the entry point of the PEP, clients will connect to the main server which will in turn connect to the endpoint. The server consists of a `pep_state` which holds the server socket, work queues and a list of tunnels. The server socket accepts new clients and reads their TLV header, if no header is present the connection will be closed. After successfully reading the TLV, a new socket is created and a connection is attempted to the endpoint specified by the TLVs.

```
struct pep_state {
    atomic_t state;

    struct pep_state_ops* ops;
    struct pep_state_work_ops* work_ops;

    struct socket* server_socket;
    struct workqueue_struct* accept_wq;
    struct workqueue_struct* forward_c2e_wq;
    struct workqueue_struct* forward_e2c_wq;
    struct work_struct accept_work;
    struct list_head tunnels;
    unsigned int total_tunnels;
};
```

Figure 4.2: PEP State Structure Code

**Tunnels** After the endpoint connection is established, the PEP will create a `tunnel`. A `tunnel` consists of 2 sockets, where data is transferred between them. Additionally to the sockets, a `tunnel` also holds the information about the sockets and the proxy connection, this includes total amount of data transferred.

**PEP Tunnel Structure**

```
struct pep_tunnel {
        unsigned int id;
        struct pep_connection client;
        struct pep_connection endpoint;
        int total_client;
        int total_endpoint;
        struct work_struct c2e;
        struct work_struct e2c;
        struct list_head list;
        struct pep_state* server;
        int state;
        int recv_callbacks;
        int packets_fowarded;
};
```

[PICTURE OF ARCHITECTURE]

### 4.3.2 Kernel Sockets

Normally an application would interact with a socket descriptors when sending and receiving data. However, inside the kernel we have access to the actual sock and socket structures. This gives us full access to all functionality and meta-data a socket can offer, such as `sk_buff` queues, callbacks and statistics like `sk_drops`. An `sk_buff`, which stands for socket buffer, is the kernel struct that holds the information about a packet. It includes both the header and data of a packet and is created when the kernel receives a packet. After that it is routed to the corresponding socket.

```
struct sock {
    atomic_t sk_drops;
    void* sk_user_data;
    ...
    struct sk_buff_head sk_error_queue;
    struct sk_buff_head sk_receive_queue;
    ...
    struct socket *sk_socket;
    ...
    void (*sk_state_change)(struct sock *sk);
    void (*sk_data_ready)(struct sock *sk);
    void (*sk_write_space)(struct sock *sk);
    void (*sk_error_report)(struct sock *sk);
    ...
};
```

Figure 4.3: Kernel Sock Structure

#### Socket Operations

**Socket Callbacks** Especially of interest is the callback function `sk_data_ready`. It is called whenever a `sk_buff` is received by a socket. That is the case for any kind of packet, not only user data but also includes the TCP handshake packets. This callback can be overwritten by a custom implementation, which is useful to detect any interactions on the socket. For example one could inspect or manipulate headers before calling the original `sk_data_ready` function. All callbacks take themselves as a parameters, thereby giving access to the socket state.

**Socket User Data** Kernel sockets allow modules to add extra information to a specific socket. The pointer `sk_user_data` can be used to point to any data defined by a module. This in combination with the callbacks

is a powerful tool for customizing socket behavior. This allows us to add additional state information to the socket.

Benefit of using kernel moduels -> direct access to socket struct instead of an fd. socket struct, user_data, data_ready callback Code snippets

### 4.3.3 Work Queues

To handle multiple concurrent events, the PEP uses `work queues`. A work in the Linux kernel is a way of handling kernel threads. They allow for a more reactive approach to threads, as a work can be queued on demand and execute simple tasks. A work is usually queued into a `work queue`. Each `work queue` represents a task and works which are queued will wait in the queue till they can be run. A task is defined as a function, to which context can be added through a `work` parameter.

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};
```

Figure 4.4: Work struct from Linux - workqueue.h

### 4.3.4 Works

A work is defined by `work_struct` (see Figure 4.4) which holds information associated with the work, most importantly the `work_func_t func`. `func` must be a pointer to a function which takes a `struct work_struct` as parameter. This is the function which will be called when a work is scheduled. The work structure should be created by the user and not be allocated on the stack as external services need access to it. Normally the work structure will be part of another structure as it is in Figure 4.2.

Listing 4.3.1: Work initialization example

```
1 void my_work_handler(struct work_struct *work);
2
3 struct work_struct my_work;
4 struct workqueue_struct * my_workqueue;
5
6 my_workqueue = create_singlethread_workqueue("my_workqueue
      ");
7 INIT_WORK(&my_work, my_work_handler);
8
9 queue_work(my_workqueue, &my_work);
```

**Work State problem** Initiating a Work poses a challenge: how to effectively track a state. In this context, 'state' refers to the status of a PEP tunnel, the sockets to utilize, and a reference to the server. A simple approach is to store the PEP server in a global variable. However, not only is this poor coding practice, but it also doesn't address the socket issue. A function that transfers between two sockets must be aware of the specific sockets to use. The work alone doesn't provide this context.

To solve this issue we can make use of a macro which Linux provides. `containerof` is a macro that retrieves a reference to the parent structure of any given struct. This in combination with the fact that we do get the original work struct as a parameter.

Listing 4.3.2: Work using containerof example

```
1 struct my_device_data {
2     struct work_struct my_work;
3     // ...
4 };
5
6 void my_work_handler(struct work_struct *work)
7 {
8    struct my_device_data * my_data;
9
10    my_data = container_of(work, struct my_device_data,
      my_work);
11    // ...
12 }
```

As described in the PEP Tunnel Structure (see above struct 4.2), the PEP has 3 main work queues:

```
struct workqueue_struct* accept_wq;
```

```
struct workqueue_struct* forward_c2e_wq;
struct workqueue_struct* forward_e2c_wq;
```

**Accept Work Queue** The main work queue for the server is the accept work queue. When a client attempts a connection to the PEP server we want to receive that notification and queue a accept work. This is an alternative to have a thread blocking on accept. The notification can be achieved by replacing the `sk_data_ready` with a custom function which checks the TCP state and queues a accept work.

Listing 4.3.3: Accept callback function

```
1  void pep_listen_data_ready(struct sock* sk)
2  {
3    struct pep_state* server;
4
5    read_lock_bh(&sk->sk_callback_lock);
6    server = sk->sk_user_data;
7
8    /* Queue accept work */
9    if(sk->sk_state == TCP_LISTEN){
10     queue_work(server->accept_wq, &server->accept_work);
11   }
12   read_unlock_bh(&sk->sk_callback_lock);
13
14   default_data_ready(sk);
15 }
```

The function `pep_listen_data_ready` (see Listing 4.3.3) outlines the process to set up accept works. We retrieve the PEP server state from the socket `sk_user_data` variable, afterwards we check the socket state for `TCP_LISTEN` which indicates that the socket is ready to accept a connection. If the socket has the correct state we queue the `accept_work` on the `accept_wq` work queue, which is part of the PEP server.

**Packet Forwarding Queues** The second two work queues handle forwarding of packets, one for each direction. The reason we use two separate work and work queues because the function has to identify which socket it should read from and which one it should send to within the tunnel socket pair.

[PICTURE PERHAPS?]

Listing 4.3.4: Forwarding callback function

```
1 void pep_client_data_ready(struct sock* sk)
2 {
3   struct pep_tunnel* tunnel = sk->sk_user_data;
4   tunnel->recv_callbacks++;
5
6   queue_work(tunnel->server->forward_c2e_wq, &tunnel->c2e)
      ;
7
8   default_data_ready(sk);
9 }
```

### 4.3.5 Kernel TCP receive and send

In the kernel, the message functions `recvmsg` and `sendmsg` are used for reading from and sending data to sockets. However, these functions bring some overhead that can lead to code clutter. To reduce this, two helper functions will be utilized instead, `pep_tcp_receive` and `pep_tcp_send`. These functions mimic the usage of the `send` and `recv` system calls, while abstracting away the complexity of the message functions.[6]

```
int pep_tcp_receive(struct socket *sock, u8* buffer, u32 size);
int pep_tcp_send(struct socket *sock, u8* buffer, u32 size);
```

The custom functions only take in the socket, buffer and size of buffer as parameters. These functions also abstract away certain error handling which are common in the kernel space such as `EAGAIN` and `ERESTARTSYS`, both of which indicate to retry a function.

## 4.4 PEP - Server

### 4.4.1 Creation

### 4.4.2 Server initialization

Server state initialization consist of creating and configuring the main server socket and work queues. The server is responsible for accepting client and creating pep tunnels. Additionally, it holds the work queues for all the PEPs functionality. Socket configuration consists of replacing the `sk_data_ready` data callback, setting `sk_user_data` to the server itself and using `setsockopt` to set both `TCP_FASTOPEN` and `TCP_NODELAY`. `TCP_NODELAY` is set to avoid socket latency by 'waiting' for larger frames.

[6] Add these functions as apendix code?

Listing 4.4.1: PEP server initialization (Simplified)

```
1  int pep_server_init(struct pep_state* server, u16 port)
2  {
3    ...
4
5    /* socket creation */
6    struct sock* sk = NULL;
7    struct sockaddr_in saddr;
8    ret = sock_create_kern(&init_net, ..., &sock);
9    if(ret){
10       printk(KERN_INFO "[PEP] init_core: Error creating
       socket\n");
11       return -EPEP_GENERIC;
12   }
13
14   ...
15   server->state = ((atomic_t){(PEP_SERVER_RUNNING)});
16
17   /* use our own data ready function */
18   write_lock_bh(&sk->sk_callback_lock);
19   sk->sk_user_data = server;
20   sk->sk_data_ready = server->callbacks->server_data_ready
       ;
21   write_unlock_bh(&sk->sk_callback_lock);
22
23   /* pep server connection info */
24   ...
25
26   pep_setsockopt(sock, TCP_FASTOPEN, 5);
27   pep_setsockopt(sock, TCP_NODELAY, 1);
28
29   ... bind and listen ...
30
31   server->accept_wq = alloc_workqueue("accept_wq",
       WQ_HIGHPRI|WQ_UNBOUND, 0);
32   server->forward_c2e_wq = alloc_workqueue("c2e_wq",
       WQ_HIGHPRI|WQ_UNBOUND, 0);
33   server->forward_e2c_wq = alloc_workqueue("e2c_wq",
       WQ_HIGHPRI|WQ_UNBOUND, 0);
34
35   ...
36
37   return 0;
38 }
```

Line 20 in Listing 4.4.1 configures the `sk_data_ready` callback to the before mentioned `pep_listen_data_ready`, when overwriting the `sk_data_ready` we need to make sure we hold the socket `sk_callback_lock` to avoid any race conditions which is done in line 17 and 21. The accept and forward work queues are allocated and created with `WQ_HIGHPRI` and `WQ_UNBOUND`.

```
WQ_HIGHPRI
    Work items of a highpri wq are queued
    to the highpri worker-pool of the target cpu.

WQ_UNBOUND
    Work items queued to an unbound wq are served
    by the special worker-pools which host workers
    which are not bound to any specific CPU.
```

Figure 4.5: https://docs.kernel.org/core-api/workqueue.html

The reason for both `WQ_HIGHPRI` and `WQ_UNBOUND` is to avoid any added latency by work queuing, especially if there are a lot of other works being queued. Work queues are used by the kernel for any deferred work, which means that there might be competition for both CPU and scheduling. The before mentioned flags assure that the PEP work's are prioritized.

Left out from Listing 4.4.1 is the creating of the accept work itself. It uses the `INIT_WORK` macro (shown in 4.3.1) with the `pep_listen_data_ready` function. After that the server is configured and ready for accept callbacks.

### 4.4.3 Accept and Endpoint connection

The accept work will call the `pep_server_accept_work` which is responsible for creating a new tunnel and connection to the desired endpoint. First the server state is fetched by using `container_of`, after which we assert that the server is in a operational state. Next, the kernel will accept a new connection in a non-blocking fashion as we know there is a incoming connection request.

Listing 4.4.2: PEP server accept function (Simplified)

```
int pep_server_accept_work(struct work_struct *work)
{
    struct pep_state* server = container_of(work, struct
    pep_state, accept_work);

    rc = kernel_accept(server->server_socket, ...);

    ... read data from socket ...

    /* Validate tlv header. */
    if(!tlv_validate(buffer)){
        return;
    }

    /* Get connect tlv options from tlv buffer */
    tlv = tlv_get_option(TLV_CONNECT, buffer);
    if(tlv == NULL || tlv->length != 6){
        sock_release(client);
        return;
    }

    endpoint = pep_endpoint_connect(tlv->optional, tlv->
    value);
    if(NULL == endpoint){
        sock_release(client);
        return;
    }

    ... configure sockets and tunnel ...

    return 0;
}
```

After successfully accepting the client we immediately allocate a buffer and read from the client. We expect it to send a TLV with TCP Fast Open, so we use the TLV library to validate and read the TLV options. Note that even if TCP Fast Open should fail the connection can still be established at the cost of the additional round trip times. Specifically we look for the `TLV_CONNECT` option, which will be used to connect to the endpoint.

### 4.4.4 Multiple Servers

Because of the callback nature of the PEP we can create multiple servers on the same host machine. By not having a 'global' server in the kernel module, we are able to potentially create as many servers as we want[7]. The program-

[7]Should I use "we want"?

mer only has to keep track of the server pointers, while the implementation of the server keeps track of the states and correct callback handling through the works. This means that each server may use the same callback function, but the state will vary.

## 4.5 PEP - Clients

### 4.5.1 Client Sockets - Endpoint Sockets

As discussed in Section 4.3.4, there are two functions responsible for forwarding packets. After queuing a forwarding work from a callback, the corresponding function is executed.

Listing 4.5.1: Client Forwarding Function (Simplified)

```
1  void pep_client_receive_work(struct work_struct *work)
2  {
3    int ret = 1;
4    int ret_forward;
5    struct pep_tunnel* tun = container_of(work, struct
       pep_tunnel, c2e);
6
7    unsigned char *buffer = kzalloc(...);
8    if (!buffer) {
9      return;
10   }
11
12   while(ret > 0){
13     ret = pep_tcp_receive(tun->client.sock, ...);
14     if(ret > 0){
15       ret_forward = pep_tcp_send(tun->endpoint.sock, ...);
16       tun->total_client += ret_forward;
17       tun->packets_fowarded++;
18     } else {
19       if(pep_tunnel_is_disconnected(tun)){
20         pep_tunnel_close(tun);
21         return;
22       }
23     }
24   kfree(buffer);
25 }
```

Listing 4.5.1 shows the function that forwards data from the client to a endpoint. First we retrieve the tunnel state by using `container_of`, this gives us the sockets which triggered the original callback. After that, a buffer is allocated and data is read from a socket and forwarded. This function will run while there is data to send, the reason is that it is more effect to read all the data that is available than to wait for a callback and work queue to

trigger.

If a socket returns 0 or less we check if the connection is closed, that is because the closing of a connection will trigger the same callback. However, when a socket is closed it will return 0 or an appropriate error code. (How shutdown is done, RW, WR, etc)

### 4.5.2 PEP Connections

By design the PEP is able to handle multiple connections at once. Each socket pair has its own work structure for each communication direction. This means, each work structure can run in parallel, in both directions. Each tunnel (socket pair) is added to a linked list in the server state. Meaning we have access to them in case we need to prematurely terminate the connections. This will avoid any memory leaks since we manage the memory for the tunnels.

### 4.5.3 Module Customization

The PEP will follow a similiar approach as congestion control when it comes to how the PEP is configured. The basic accept and forward functions will be defined by a table, which can be created by any future model. Each server has a pointer to this table, which it uses when creating works. This mimics the way we configure socket callbacks and makes each PEP server more customizable. Each individual PEP server can have different forward functions, or keep the orignal ones.

```
struct pep_state_work_ops {
    void (*accept)(struct work_struct *work);
    void (*forward_c2e)(struct work_struct *work);
    void (*forward_e2c)(struct work_struct *work);
};
```

Figure 4.6: Work operation table

The work function table in combination with the fact that the PEP supports multiple servers, means that each indiviudal server can be configured different without needing to change any of the original code. Simply creating a new `struct pep_state_work_ops` and supplying new work functions is enough:

[EXAMPLE CODE][8]

```
struct pep_socket_callbacks {
    void (*server_data_ready)(struct sock* sk);
    void (*client_data_ready)(struct sock* sk);
    void (*endpoint_data_ready)(struct sock* sk);
};
```

Figure 4.7: Callback function table

### 4.5.4 System Configurations

The PEP will require some configuration outside of the kernel module itself. Linux uses `sysctl` for system configuration. Most importantly we want to enable TCP Fast Open and IP forwarding, IP forwarding will allow the Linux machine to act as a router and forward packets, which is important as the PEP will handle all other traffic as well. Both these options are under `net.ipv4` ...

```
$ sysctl -w net.ipv4.tcp_fastopen=3
$ sysctl -w net.ipv4.ip_forward=1
```

**Buffer sizes** The PEP works by buffering as much data as possible on the sockets themselves. This way we avoid buffering interactive traffic that simply passes by. The size of a socket's buffer can be configured with `setsockopt`, however this is not reliable and has to be done for each socket. Instead we can configure the socket buffer sizes system-wide. Under `net.core` there exists configurations for overall receive buffer sizes: `rmem_max` and `wmem_max`. There also exists the same for the default variables `rmem_default` and `wmem_default`.

```
$ sudo sysctl -w net.core.rmem_max=<size>;
$ sudo sysctl -w net.core.wmem_max=<size>;
$ sudo sysctl -w net.core.rmem_default=<size>;
$ sudo sysctl -w net.core.wmem_default=<size>;
```

Additionally, under `net.ipv4` there are options to configure the amount of memory in bytes a TCP socket can buffer for both total, read and write. Each contains three numbers: the minimum, default, and maximum values.

```
$ sudo sysctl -w net.ipv4.tcp_rmem='<min size> <size> < max size>';
$ sudo sysctl -w net.ipv4.tcp_wmem='<min size> <size> < max size>';
$ sudo sysctl -w net.ipv4.tcp_mem='<min size> <size> < max size>';
```

[8] Talk about callback table too

### 4.5.5 Userspace?

Could this be implemented in user-space?

### 4.5.6 Threads Vs. Callbacks

Small discussion around that?

### 4.5.7 Using Netfilter

Forwarding using Netfilters, why we dont want this.

## 4.6 Memory

Memory Management in C, why its important, how we deal with it.

# Chapter 5

# Evaluation

The evaluation of the PEP wil

## 5.1 Initial configuration

Sender –> Receiver –> Receiver
Data being pushed from Sender to receiver, file size, speed, delay.

### 5.1.1 Traffic Control Options

Linux has support for network interface configurations using the TC (traffic control) command. TC allows the configuration of packet scheduler, bandwidth, delay and jitter etc. These options combined with the fact that each network interface can have it's own configuration, allows for very precise testing environments.
fq codel does not allow the configuration of delay, this means we have to configure the delay on the path of the ACKS? We have to configure the delay on sender facing interface cards. [1]

### 5.1.2 Scheduling Algorithms

The choice of scheduling algorithm is very important for our test scenarios. As it will greatly affect the results of our tests, we will compare some against each other. The main goal is to highlight the effect and find the optimal algorithms for our PEP.

**FIFO** In our test case FIFO will have the effect of creating up a queue at our PEP. This is detrimental for the interactive UDP flow as it will be stuck in the queue, building up delay. This behavior can be explained by the fact that the non-interactive flow has a much higher sending rate than the

[1] Picture of testbed setup

interactive flow, thereby quickly filling the finite queue with non-interactive packets.

**FQ CoDel: IMPORTANT** FQ CoDel has a solution for this problem by providing a fair queuing mechanism.

**PFIFO** Another alternative is Priority FIFO (PFIFO). As the name suggests it combines the concepts of a basic FIFO queue with priorities. This can reduce delay

### 5.1.3 Interactive vs PEP

Our first experiment consists of having a interactive flow (100 byte UDP packets at 2kBps) competing with a file transfer, one default end to end and one through our PEP. Highlighting some of the initial differences between an end to end connection and a PEP.

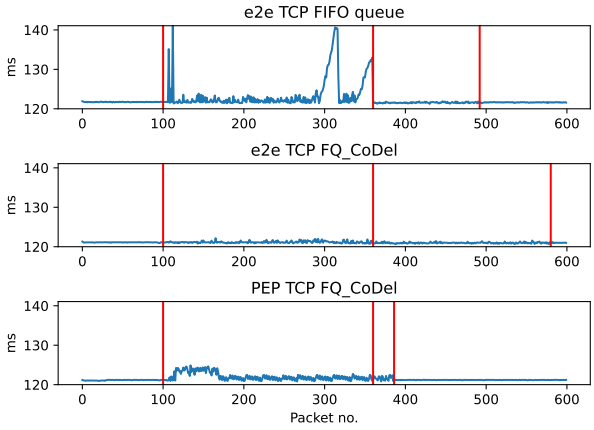


Figure 5.1: Interactive UDP traffic

### 5.1.4 PEP vs E2E tests

The first test consists of evaluating our PEP against a default TCP end to end (E2E) connection, while also highlighting the difference a packet scheduler can make. Fig. 5.1 shows the results, the red lines represent important events in the time-line. The first line represents the start of a file transfer, the second shows a bandwidth change from 10mbit to 75mbit, while the last line shows when the file transfer finished.

The first time-line shows an end to end TCP transfer using a FIFO queue, in this case BFIFO.

On the second timeline we see the behavior of an end to end connection using FQ CoDel. MORE The last timeline shows the PEP using FQ CoDel. Looking at the graph we can see two important differences between the default end to end TCP and the PEP. Firstly, the PEP has higher latency fluctuations than the E2E connection, shown by the blue liens on the graph.

### 5.1.5 10x10 tests

To further evaluate the PEP we conducted an experiment where we have 1-10 flows competing, once using the PEP and once end to end. The goal is to see how competing flows affect the behavior of or PEP

### 5.1.6 Spike

# Chapter 6

# Conclusion

## 6.1 Future Work

# Bibliography


[1] S. K. Agrawal and Kapil Sharma. 5g millimeter wave (mmwave) communications. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, page 3630–3634, Mar 2016.

[2] Olivier Bonaventure, Mohamed Boucadair, Sri Gundavelli, SungHoon Seo, and Benjamin Hesmans. 0-RTT TCP Convert Protocol. RFC 8803, July 2020.

[3] David Borman. TCP Options and Maximum Segment Size (MSS). RFC 6691, July 2012.

[4] Kristjon Ciko, Michael Welzl, and Peyman Teymoori. Pep-dna: A performance enhancing proxy for deploying network architectures. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–6, 2021.

[5] J.-M. De Goyeneche and E.A.F. De Sousa. Loadable kernel modules. *IEEE Software*, 16(1):65–71, 1999.

[6] Wesley Eddy. Transmission control protocol (tcp). Request for Comments RFC 9293, Internet Engineering Task Force, Aug 2022.

[7] Jim Griner, John Border, Markku Kojo, Zach D. Shelby, and Gabriel Montenegro. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[8] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, page 181–194, Berlin Germany, Nov 2011. ACM.

[9] Toke Høiland-Jørgensen, Paul McKenney, dave.taht@gmail.com, Jim Gettys, and Eric Dumazet. *The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm.* Number RFC 8290. Jan 2018.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[11] Alberto Medina, Mark Allman, and Sally Floyd. Measuring interactions between transport protocols and middleboxes. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, page 336–341, Taormina Sicily, Italy, Oct 2004. ACM.

[12] Michele Polese, Marco Mezzavilla, Menglei Zhang, Jing Zhu, Sundeep Rangan, Shivendra Panwar, and Michele Zorzi. milliproxy: A tcp proxy architecture for 5g mmwave cellular systems. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 951–957, 2017.

[13] Cristian García Ruiz, Antonio Pascual-Iserte, and Olga Muñoz. Analysis of blocking in mmwave cellular systems: Application to relay positioning. *IEEE Transactions on Communications*, 69(2):1329–1342, Feb 2021.

[14] W. V. Wathsala, Buddhika Siddhisena, and Ajantha S. Athukorale. Next generation proxy servers. In *2008 10th International Conference on Advanced Communication Technology*, volume 3, pages 2183–2187, 2008.

[15] Michael Welzl. Network congestion control: Managing internet traffic. *Network Congestion Control: Managing Internet Traffic*, pages 1–263, 05 2006.

[16] Michael Welzl, Dimitri Papadimitriou, Bob Briscoe, Michael Scharf, and Michael Welzl. Open Research Issues in Internet Congestion Control. RFC 6077, February 2011.