DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# Design and Evaluation of a Predictable Interface for a User Space IP Stack

Oliver Layer

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

# Design and Evaluation of a Predictable Interface for a User Space IP Stack

# Entwurf und Evaluation einer vorhersehbaren Schnittstelle für einen IP Stack im Benutzermodus

| | |
|---|---|
| Author: | Oliver Layer |
| Supervisor: | Prof. Dr. Marco Caccamo |
| Advisor: | Dr. Alexander Züpke, Dr. Andrea Bastoni |
| Submission Date: | 16.05.2022 |

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, Oliver Layer

# Abstract

An issue for real-time systems is to predictably interact with the network stack. This is especially the case when accessing the network stack simultaneously from multiple applications running on the operating system. The interface connecting the applications to the network stack has to focus on predictability to ensure the real-time property of applications is not violated, while also respecting usability concerns by for example offering an ergonomic socket API.

Usually, the network stack runs in kernel space (e.g. for *Linux*), but to facilitate security and support for embedded systems that are based on a microkernel approach, we let the network stack run in user space.

In this setting, this thesis contributes an implementation of an interface for a user space network stack that aims to solve some of the predictability and usability issues found in currently available interfaces for user space network stacks. The interface has been developed as a prototype running on Linux. Furthermore, this thesis selects a suitable network stack to develop the interface for, gives reasoning for the design decisions of the interface, outlines the implementation of the interface and finally evaluates the interface using practical as well as theoretical methods.

The results of the evaluation show that the implemented interface has a low communication latency with an average of 13µs. The interface outperforms other existing interfaces by having up to *20x* lower mean latencies when more than eight sockets are communicating in parallel. For a lower number of parallel communicating sockets, it performs similar to existing interfaces. Moreover, it was assessed that the predictability mechanisms work in practice.

Even though the interface is a prototype and only supports TCP sockets, the findings regarding the design and implementation of the interface can be used as a starting point to enhance the communication between applications and the network stack of real-time systems.

# Contents

# Acronyms

**AIO** Asynchronous I/O

**API** Application Programming Interface

**BSD** Berkeley Software Distribution

**DHCP** Dynamic Host Configuration Protocol

**DNS** Domain Name System

**DoS** Denial of Service

**DPDK** Data Plane Development Kit

**FIFO** First In First Out

**futex** fast userspace mutual exclusion

**HPC** High-Performance Computing

**I/O** Input/Output

**IP** Internet Protocol

**IPC** Inter-Process Communication

**NIC** Network Interface Card

**OS** Operating System

**pifus** Predictable Interface for a User Space IP Stack

**QoS** Quality of Service

**TAP** Network tap

**TCP** Transmission Control Protocol

**UDP** User Datagram Protocol

# 1. Introduction

## 1.1. Motivation

Real-time systems demand applications and their functionalities to be *predictable*. This property is necessary to make sure that deadlines are held. Nowadays, these systems are interconnected and must therefore have their own network stack.

Generally, to facilitate verification, applications of real-time systems must ideally be independent and small. This also holds true for the network stack of these systems, which means that the stack may run in user space, decoupled from the operating system kernel. Running in user space also gives certain advantages, for example in terms of security, and fits to embedded systems using a microkernel approach.

An open question in this context is how application processes can interact with the network stack in a *predictable* way. In this context and in the context of this thesis, *predictability* means that real-time applications should have precedence over non real-time applications when accessing the network stack. Moreover, one ideally wants to minimize the communication latency to the network stack in general.

Furthermore, *predictability* means that the uncertainty of synchronous I/O multiplexing should be reduced to a minimum. The uncertainty on the one hand originates from having to wait until the I/O is ready to execute operations and on the other hand by the search made in linear time that is for example performed by `select`.

There are several open problems when designing such an interface for accessing the network stack. One of these problems is how to deal with concurrent access of applications to the network stack. In general, the communication between the stack and the applications is a problem to be solved.

Another problem arises when thinking about how to ensure *predictability*, so that real-time applications have precedence over normal applications. The objective is to prevent non real-time applications from interfering with real-time applications when accessing the same network stack.

Moreover, a problem is how to mitigate the uncertainty of synchronous I/O multiplexing.

## 1.2. Existing solutions

Possible solutions to the problems mentioned before are tackled in different areas. For example, efficient communication between applications is a problem that operating systems with microkernels try to solve. Further existing solutions to these problems are outlined in Chapter 2.

In microkernels, functionality of the operating system is distributed across several applications instead of a single monolithic kernel. This concept requires the applications to communicate with each other to fulfill their tasks.

Another problem where solutions already exist is the problem of the uncertainty of synchronous multiplexing I/O. Some operating systems, such as Solaris or Windows, use completion ports as I/O model to mitigate the issues that come with synchronous multiplexing I/O. Completion ports are an I/O model that is asynchronous, event-based and centered around the aspect of notifying applications when an I/O operation is complete. This mitigates the issue to have to wait on sockets to become ready, as it is the case for synchronous multiplexing I/O.

Therefore, when considering each single problem in isolation, single ad-hoc solutions exist. The issue still persisting is to bring these solutions together and build an interface for a user space network stack that combines the solutions of all the single problems in the areas of predictability and communication. This would fill the gap in research.

## 1.3. Contributions

We present *pifus*, an acronym for a **P**redictable **I**nterface **F**or a **U**ser space IP **S**tack.

*pifus* is an interface written in C that solves the problem of predictably connecting multiple applications to a user space network stack, filling the gap previously mentioned. *pifus* is the main contribution of this thesis. It leverages on shared memory and especially on futexes for blocking synchronization. Shared memory as well as futexes are the foundation of the communication paradigms of *pifus*. A prototype of *pifus* was developed on Linux. While Linux is not a microkernel, it is an operating system well suited for rapid prototyping. Furthermore, shared memory and futexes are available on Linux and microkernels, so the general approach presented in *pifus* is applicable to a wide range of systems.

*pifus* is a library that can be used by any application. It uses a shared memory region per application and further shared memory regions for every socket that is opened. These regions are used to communicate with the *pifus* backend which is integrated into the network stack. *pifus* enables multiple applications to concurrently access the network stack through the interface.

This thesis gives insight into central concepts that were elaborated when *pifus* was developed. These concepts can be taken into consideration by the research community when developing similar interfaces.

Another contribution of this thesis is the evaluation of *pifus*. It assesses *pifus* capabilities and compares it to the netconn interface of the lwIP network stack.

Furthermore, a contribution is made by presenting and comparing different user space network stacks.

## 1.4. Outline

Chapter 2 deals with the background and related work of the thesis. The background part elaborates on technologies relevant for the thesis, while related work summarizes academic publications that are related to this thesis.

In contrast, Chapter 3 is about selecting the most suitable user space network stack to develop an interface for. It defines criteria, presents several stacks and finally rates each stack according to the identified criteria.

Chapter 4 defines what is expected from the interface. It outlines several requirements that the interface should fulfill.

Chapter 5 describes how *pifus* has been designed and implemented. It also presents alternative designs.

Differently, Chapter 6 evaluates *pifus* using practical and theoretical methods.

In the end, Chapter 7 draws a conclusion and shows limitations and possible future work that could be done in context of the thesis.

# 2. Background and Related Work

This Chapter describes the background of *pifus* by providing an explanation to underlying and comparable concepts in Section 2.1. Furthermore, related work is presented in Section 2.2.

## 2.1. Background

Background about network stacks running in user space is presented in Subsection 2.1.1, while general information on multiplexing I/O can be found in Subsection 2.1.2. Moreover, an explanation of completion ports is provided in Subsection 2.1.3.

### 2.1.1. User space network stacks

User space network stacks are used in areas such as in microkernel based operating systems or in High-Performance Computing (HPC) applications. The main reason to use a user space network stack is to circumvent the overhead induced by system calls into the operating systems kernel.

Generally, with the rise of frameworks such as DPDK, moving the network stack to user space becomes more common. DPDK offers an interface to communicate with the NIC in a simple way.

Especially for applications in the HPC area, kernel bypass is relevant because their main goal is performance. Using a user space network stack is also an option for reducing latencies and therefore improving predictability due to the bypass of system calls.

There exist multiple user space network stacks. This thesis presents some of them and based on criteria selects a network stack that should be used with *pifus*. This process is part of Chapter 3.

### 2.1.2. Multiplexing I/O

I/O multiplexing is essential when programming network applications that use multiple sockets. With I/O multiplexing, applications can be blocked and be notified upon any event that occurs on one of the sockets.

Classic multiplexing I/O system calls are `select` [1], `poll` [2] and `epoll` [3]. Conceptually, these functions take a list of file descriptors to watch and then block the thread that invoked them and return when there was any activity on one or multiple of these file descriptors or a timeout has passed, if any was given.

When taking a closer look at the functions, they differ in their implementation. `select` only supports 1024 sockets due to internally using a bitmap. Furthermore, `select` has $O(n)$ time complexity (with $n$ being the number of supplied file descriptors) because it iterates over all supplied file descriptors to check for new events. In terms of usability, `select` only returns the number of file descriptors that became ready, making it necessary for the developer to further iterate over the list that was initially supplied to and then modified by `select` to check which file descriptor had activity [1].

In contrary to `select`, `poll` [2] has no restrictions on the amount of file descriptors. Regarding time complexity, there is no difference to `select`.

The latest addition to these algorithms, `epoll`, instead has a time complexity of $O(1)$ for waiting. `epoll` shifts the costly part to the creation of the data structure when the developer declares interest in certain file descriptors. The creation of the data structure still has a time complexity of $O(n)$, but once created, the data structure can be hold for the lifetime of a program execution. The operating system modifies the data structure when file descriptors are woken, putting these into a separate list in the data structure. When the developer then blocks using `epoll_wait`, the call just checks the list and may return immediately if the list is not empty. In this way, the runtime of `epoll_wait` is constant.

Nevertheless, synchronous I/O multiplexing as presented in this Section still has its problems, even though `epoll` performs better than `select` or `poll`. One of these problems is the *thundering herd problem*. The problem describes the issue of the kernel having to wake up all threads waiting on a shared file descriptor at the same time, even though only one of these threads is able to handle the event [4]. Furthermore, general problems when implementing this concept are for example how to save different interests in same file descriptors for different threads (which influences space complexity) or how to organize the queues indicating that certain file descriptors have become ready (which influences time complexity). For instance, the queues indicating file descriptor activity could be stored on a per-thread basis and would then have to be iterated over after to infer which file descriptor had activity.

These are some reasons that brought to the evolution of alternatives to synchronous multiplexing I/O or in general I/O based on a readiness model. One of these alternatives and inherent motivation for this thesis are completion ports, which are presented in the next Subsection.

### 2.1.3. Completion ports

Completion ports are an asynchronous I/O model that mitigates the issues presented in Section 2.1.2 by overcoming the concept of waiting for the readiness of a file descriptor with using asynchronous notifications instead of synchronous blocking. With completion ports, operations can be queued and a notification is sent when the operations are completed. Therefore, a readiness indicator for sockets such as `epoll` is not needed, as operations can be queued and will only be executed when it is possible for them to be executed.

#### Readiness model vs. Completion model

There are different ways to do asynchronous I/O. Instead of using a readiness based I/O model, completion ports are using a completion based I/O model, as indicated by the name.

The difference between a completion based I/O model and an I/O model based on readiness is that in the first case, the developer expresses an intent to do a certain I/O operation with certain arguments without blocking the thread and then gets notified when the operation has *completed*. On the other hand, in the latter case, the developer is being told when the I/O operation is *ready* to be executed, and then the developer explicitly has to execute the operation with the desired arguments.

For example, the flow of events when trying to `read` from a socket in case of a readiness I/O model would look like this:

1. The developer sends intent to `read` from a socket
2. The API tells the developer that the socket is *ready* for the `read`
3. The developer allocates buffers for the data and executes `read` on the socket
4. The buffer is filled

In contrary, for completion ports, the flow of events would look like this:

1. The developer allocates buffers for the data and executes `read` on the socket
2. The buffer is filled and the API tells the developer that the operation has been *completed*

One can see that the interaction with the API is easier when using completion ports than it is when using a readiness I/O model.

**Implementations**

Completion ports are supported by several operating systems, such as Windows, where they are called *I/O completion ports* (IOCP), or Solaris. Furthermore, `io_uring` [5] adds I/O completion ports to Linux. Being open source, the implementation of `io_uring` is analyzed next.

`io_uring` maps a submission and completion queue (both are ring buffers) into user space using `mmap`. Then, the application can add an entry in the submission ring buffer. An entry allows to specify the type of the operation, a file descriptor, pointer to buffers and more, depending on the operation that will be executed. After the operation has been processed by the Linux kernel, a resulting entry will be added to the completion queue and the application can dequeue this entry. `io_uring` offers a poll mode, where no system call is needed to execute I/O calls. When not using the poll mode, a system call has to be used to tell the kernel that new operations have been submitted. The interaction between the kernel and the application using the queues can be seen in Figure 2.1. Due to reducing the amount of system calls needed for I/O, `io_uring` claims to have better performance than traditional synchronous I/O.

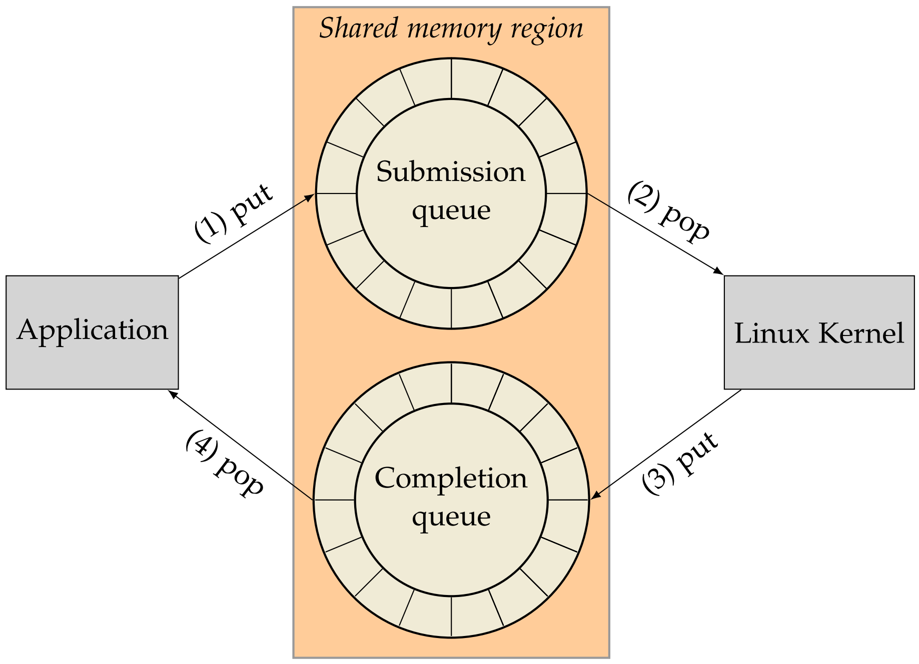

Figure 2.1.: Interaction between the kernel and the application when using `io_uring`.

Another way of using completion ports in Linux is the POSIX AIO [6] interface.

Due to several flaws in the implementation, POSIX AIO is not widely used. For example, the `glibc` implementation uses a user space thread pool which essentially performs blocking I/O inside the thread pool [7], which decreases performance when executing many I/O operations due to threads sleeping and being woken up repeatedly. Furthermore, POSIX AIO only supports signals and threads as completion indicator, of which both are not suitable for a large amount of I/O operations due to performance issues [7].

The IOCPs of Windows also use a thread pool with a configurable amount of threads to execute I/O operations.

## 2.2. Related Work

Providing an interface for a network stack is a common task when developing microkernels. Microkernels face the same challenge of coordinating multiple applications that want to access the network stack. In this area, there is research work that has similarities to this thesis.

For example, [8] divides the network stack itself into smaller user space processes, so that protocols, such as TCP or IP, execute inside their own process [8, Figure 1]. While using modern multicore hardware, it aims at showing that such an architecture may be suitable to compete with monolithic approaches in terms of performance [8, Abstract]. It uses shared memory as IPC mechanism with lock-free data structures consisting of queues for requests, pools for data exchange and a database for e.g. associating requests with data [8, Fast-path channels]. One of the first publications to propose such an architecture was [9].

The main difference between *pifus* and these approaches is that they split the network stack into smaller pieces, while *pifus* tries to keep the network stack and its protocols running in one process.

An approach that does not split the network stack into smaller pieces, but uses a single process stack design, is *IsoStack* [10]. *IsoStack* aims to provide lock-free, event-driven and interrupt-free execution of the network stack [10, IsoStack Architecture].

Queues in shared memory are used for communication between the stack and the applications. To avoid performance issues due to contention, one notification queue is created per logical processor. The notification queues are solely used for notifying that the application has submitted new commands, but do not include the commands themselves. This has the advantage that the amount of notification queues is constant and does not scale with the number of applications that are using the stack, which in turn makes multiplexing and polling easier for the receiving side. The actual request and response queues are contained in shared memory and are application-specific [10, Message Queues]. *IsoStack* further implements a round-robin scheduling to determine which queue to work on and consequently does not provide any QoS guarantees, such as priorities on sockets. [10, Conclusions and Future Work]

Another approach that was influenced from the approach that microkernels take is the user space networking system *Snap* [11]. Amongst other things, a main goal of *Snap* is to increase the network performance, especially tailored to communication inside data centers [11, Abstract].

*Snap* is essentially a user space process, which directly accesses the NIC and communicates with other user space processes through shared memory with lock-free data structures [11, Figure 1]. The communication with other processes happens over the so-called *Pony Express API*, which uses Unix domain sockets to advertise shared memory regions, in which a submission queue, completion queue and further space for the payloads reside [11, Pony Express: A Snap Transport]. *Snap* uses a completion I/O model.

*Snap* also supports different features regarding the lower end of the stack, such as NIC offloading [11, Introduction]. Most importantly, *Snap* itself does not implement TCP as transport protocol, but rather its own network protocol called *Pony* [11, Pony Express: A Snap Transport]. Therefore, in terms of functionality, it is a special purpose network stack and not a general one like e.g. the Linux network stack.

All the presented approaches do not target real-time systems and are therefore not using a network stack or communication paradigms that explicitly address predictability concerns. Instead, many approaches address performance concerns rather than predictability concerns.

Nevertheless, the communication patterns used between the processes in the approaches explained above are also valid considerations for this thesis. Especially the design of *Snap* has to be taken into consideration, as it offers an interface with an I/O concept similar to completion ports.

*pifus* wants to fill the gap in research that exists by providing an interface that makes use of a user space network stack and completion ports while having predictability as main concern.

# 3. IP stack selection

Developing an interface for a user space IP stack requires a thoughtful selection of the network stack to use. This chapter describes thoughts about this selection process, while the subsequent chapter describes requirements of the interface to the network stack that is selected here.

Section 3.1 describes the criteria that the network stack should fulfill to be selected, while Section 3.2 evaluates different user space stacks according to the criteria. Section 3.3 then presents the decision on which stack was selected.

## 3.1. Criteria

The network stack should fulfill certain criteria. One point is the *maturity* of the network stack. As this interface is not only a prototype, but may be evolved into a production-ready interface. Therefore, a network stack is needed that could also be used in a production environment. Furthermore, this interface targets real-time systems, which require mature network stacks.

Another point is the *portability* of the network stack. The network stack and the interface should also be portable to operating systems other than Linux. Ideally, it should also be portable to constrained devices such as embedded systems.

Furthermore, the network stack should be a *complete* network stack, meaning it should ideally support TCP, UDP and optionally even more protocols or a more direct access to the network, such as raw sockets.

In addition, the performance of the network stack is also an important criterion. It should be possible to saturate modern Ethernet links with the network stack.

Lastly, the network stack should ideally offer a fine-grained API, which allows controlling the network stack on a detailed level. For instance, the memory management should be configurable and access to low-level TCP features should be given, e.g. disabling *Nagle's algorithm*.

## 3.2. Stacks

### 3.2.1. lwIP

lwIP, standing for lightweight IP, is a user space network stack initially developed by Adam Dunkels [12]. lwIP targets embedded systems, due to its focus on low code size and low memory usage. It is modular, allowing to disable features of the stack, resulting in smaller binary sizes if done so.

lwIP is a complete IP stack and supports IPv4/6, TCP, UDP and application level protocols, such as DNS or DHCP [13].

The combination of modularity and resource-constrainedness makes lwIP a popular choice as network stack for embedded systems. For example, lwIP is used in the real-time operating system FreeRTOS [14].

Furthermore, there is a variety of drivers for lwIP so that it can be used with several NICs, most of them designed for the embedded field [15]. There is also a port for Linux, allowing to use the TAP interface.

The lwIP main stack logic is not thread-safe and therefore requires synchronization when used from multiple threads.

lwIP offers three different APIs [16] [17], with each having a different level of abstraction:

- raw API
- netconn API
- socket API

The raw API is a low-level and event-driven API. The netconn and the socket APIs are implemented using the raw API. Of the three APIs, the raw API therefore offers the most direct interface to the lwIP stack. Generally, the raw API works with callbacks. The user of the API can execute an operation on the connection and for operations that may take longer, a callback is invoked after they have been completed. Furthermore, the raw API offers features such as zero-copy and promises to deliver "[...] maximum performance and minimum memory footprint" [18].

The netconn API on the other hand is on a higher level of abstraction. In contrast to the raw API, the netconn API blocks when executing operations on a connection. To be able to use multiple sockets simultaneously, it makes use of a certain thread model, where the netconn connection running on a thread communicates with the lwIP stack through mailboxes (see [12, Section 5]). See the right-hand side of Figure 3.1 for a visualization of the netconn API. Netconn also offers a non-blocking mode, in which the callbacks are invoked by the lwIP main thread [19].

The socket API uses the netconn API to implement an interface mimicking BSD sockets.

lwIP also offers the possibility to specify a threading mode. In the so called OS mode, the main logic of lwIP is executed in the `tcpip-thread`, which is completely managed by lwIP. In the mainloop mode, the stack logic must be executed by the user of the lwIP stack by regularly invoking lwIP. When using the mainloop mode, only the raw API is available. A visualization of the raw API in combination with the mainloop mode can be seen in Figure 3.1 on the left-hand side.

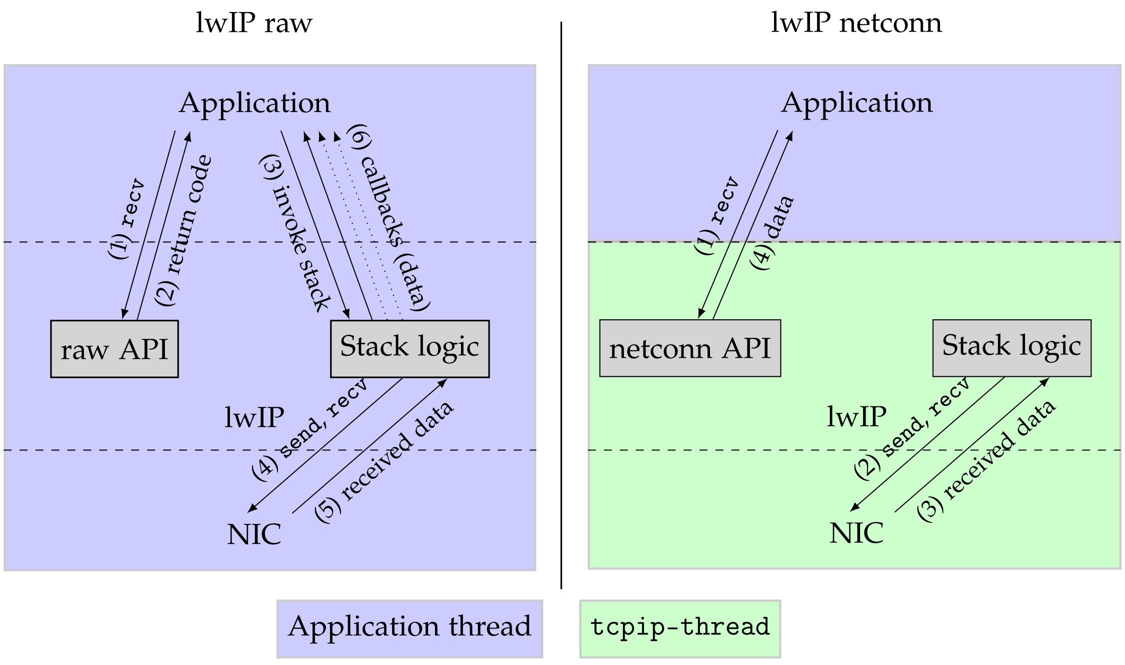

Figure 3.1.: Comparison between the lwIP raw API in mainloop mode and the lwIP netconn API. The numbers denote the temporal order.

### 3.2.2. mTCP

Another user space IP stack is called mTCP [20]. mTCP is not a complete IP stack, but only supports basic IP functionality and TCP. mTCP is a research stack, designed for multicore systems to achieve high performance rather than having a low footprint [20, Abstract].

mTCP itself is not well maintained and not used in any production systems. Several GitHub issues [21], addressing main concerns about that stack while being unanswered for several years, indicate that the stack is immature. There is no recent activity in issue solving or new commits to the stack.

The main advantage of mTCP is its design towards high performance. To achieve this, mTCP uses packet-level and socket-level batching [20, Introduction] and lock-free data structures [20, Design].

mTCP only supports a single application using the stack [20, Design]. It offers an API inspired by BSD sockets and another `epoll`-like event-driven API and is based on DPDK.

The structure of mTCP can be seen in Figure 3.2.

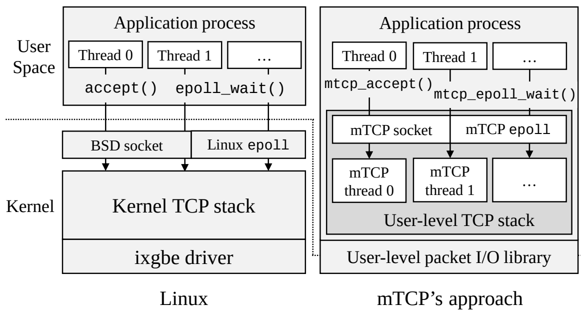

Figure 3.2.: mTCP's architecture compared to the Linux IP stack [20, Figure 3].

### 3.2.3. PicoTCP

PicoTCP is a complete network stack, supporting IPv4/6, UDP, TCP and other application level protocols [22]. Similar to lwIP, it has a small footprint, is modular and designed to be portable. It targets embedded systems and is intended for production use.

PicoTCP is maintained by Altran and the open-source community, with the latest release date being in 2017 [23].

There are several ports available, such as ports to FreeRTOS, mbed-RTOS, Linux or Windows. There are drivers available especially for embedded NICs.

PicoTCP offers two APIs, namely an event based API and an API similar to BSD sockets.

Generally, PicoTCP runs the stack logic on a single thread, while only supporting interaction with the stack from multiple threads for a subset of functions. (e.g. only reading from sockets)

### 3.2.4. F-Stack

F-Stack is intended to be a complete IP stack, but has no support for IPv6 yet. It was designed to be a high performance stack for cloud use-cases [24].

It does not require separate drivers written for it, but rather uses DPDK as a backend. F-Stack therefore supports every NIC and platform that is also supported by DPDK. The ARM architecture is not supported by F-Stack.

It builds upon the FreeBSD stack and offers an interface inspired by `kqueue` and `epoll`.

### 3.2.5. smolTCP

smolTCP is a complete IP stack, offering the possibility to use IPv4/6, UDP, TCP and other application level protocols. Some niche features of the IP protocol are not yet supported, such as certain IPv4 options [25].

The goal of smolTCP is to offer a simple, robust user space network stack that can be run on bare-metal and real-time systems [25].

It is maintained by the open-source community and used in some open-source projects, such as *RedoxOS* or *RustyHermit*, of which the latter is a lightweight unikernel which previously used lwIP [26].

smolTCP tries to be an alternative to lwIP by using Rust as a modern counterpart to C. It does not use any heap allocation at all and claims to have a good performance, saturating Gigabit Ethernet links [25].

Currently, there is only a single driver for a specific NIC. Therefore, usage with real world hardware is limited. Nevertheless, smolTCP offers integration with virtual interfaces, such as TAP.

The smolTCP API is inspired by BSD sockets, with an addition called `waker` that allow for event based interaction. The buffer management is done by the user of the API.

## 3.3. Result

All previously presented stacks were compared to each other using the criteria defined in Section 3.1. The results are shown in Table 3.1.

| Stack | Maturity | Portability | Completeness | Performance | API |
|---|---|---|---|---|---|
| lwIP | ✓ | ✓ | ✓ | ~ | ✓ |
| mTCP | X | ~ | X | ✓ | ~ |
| PicoTCP | ✓ | ✓ | ✓ | ~ | ~ |
| F-Stack | ✓ | X | ~ | ✓ | ~ |
| smolTCP | ~ | ~ | ~ | ✓ | ~ |

Table 3.1.: Comparison of the network stacks.

As seen in the Table, lwIP, PicoTCP and F-Stack are the most mature stacks in the comparison. smolTCP was not rated as mature, as it is a rather new stack, constantly being developed. mTCP was rated worst in this category, as the stack is intended for research, but not production, and is unmaintained since several years while having many open issues.

Regarding portability, lwIP and PicoTCP scored the best among the stacks. They are both designed for portability and there exists a pool of selectable drivers. In contrary, F-Stack has limitations when it comes to CPU architectures other than x86. mTCP uses DPDK and therefore supports several systems, but is stuck at an old version of it. For smolTCP there currently exists only one driver for a NIC.

When it comes to completeness, lwIP and PicoTCP perform better than the other evaluated stacks. They both offer complete support for standard network protocols such as IP, UDP and TCP and additionally support further application level protocols, such as DNS. While smolTCP and F-Stack support UDP and TCP, mTCP only supports TCP.

In terms of performance, mTCP, F-Stack and smolTCP have an edge over lwIP and PicoTCP. mTCP and F-Stack are designed considering performance as first priority, while for lwIP and PicoTCP, portability is the main concern. smolTCP benchmarks show good performance [25].

Considering the API, lwIP has the advantage of offering three APIs with different abstraction levels, of which one allows full control over the network stack. The other stacks offer only one or two APIs. For some stacks, only a BSD socket like-API is offered, which does not allow much control over the stack.

For the above stated reasons, lwIP was chosen as the network stack to design and evaluate an interface for. Especially the maturity and the offered APIs have contributed to this decision. Furthermore, predictability rather than performance is the main priority of *pifus* as seen in the subsequent Chapter.

# 4. Analysis

When developing the interface for a network stack running in user space, the main problem statement is the question of how processes communicate with the network stack *asynchronously*, *scalably* and *predictably*.

*Asynchronicity* means that operations on the network stack can be executed by the user without blocking the thread. Asynchronicity is part of the usability requirements and further explained in Section 4.1.

*Scalability* indicates that the usage of the network stack is possible for multiple concurrent processes without impacting performance. The requirements on scalability are elaborated on in Section 4.2.

*Predictability* on the other hand means that the latencies of executing operations are foreseeable and do not jitter. Furthermore, they should ideally be independent of input sizes, e.g. the number of operations. Requirements on predictability are explained in Section 4.3.

Other requirements are *functionality* (see Section 4.4), *security* (see Section 4.5) and *further non-functional requirements* (see Section 4.6).

## 4.1. Usability

The interface should offer an easy to use API. Easy to use means that the API should be similar to existing socket APIs such as BSD sockets, so that existing applications may be migrated easily and the semantics of the API is clear to the user at first glance.

While the simplicity is one point, another point is the general concept of the API. While most socket APIs are synchronous, this interface should offer an asynchronous way of interacting with the network stack, similar to `io_uring` or Windows `IOCP`. This means that a completion I/O model instead of a readiness I/O model should be used.

Furthermore, the usage of the API should be possible in a plug and play way. This means that the user should not have to configure anything prior to using the interface.

The API itself should exist in form of a library which can be used in an arbitrary application.

## 4.2. Scalability

The interface should be scalable, this means that it should support the concurrent execution of operations on the network stack originating from different processes. Scalability further means that concurrent usage of the interface from different application processes should not severely impact the performance.

Moreover, scalability implies that the interference between different applications using the interface should be minimized. Interference between the application processes can lead to worse scalability and predictability, e.g. when global coordination in the form of locks between the applications is needed. As this also concerns predictability, see also the explanation in Section 4.3.4.

## 4.3. Predictability

Predictability is the most important requirement of the interface. Predictability has several aspects, such as scheduling (see Subsection 4.3.1), latency (see Subsection 4.3.2), fault tolerance (see Subsection 4.3.3) and interference avoidance (see Subsection 4.3.4). Attention has also to be paid to the problem of application multiplexing in regard to the predictability (see Subsection 4.3.5).

### 4.3.1. Scheduling

The interface should offer the possibility to categorize sockets according to their time criticality similar to QoS mechanisms offered by some network stacks. Operations of a socket with a high time criticality should always be preferred over operations of a socket with a lower time criticality. This means, some sort of socket priorities have to be implemented. In the interface, it should be possible to set a priority on a certain socket. The priority levels should be discrete, e.g. in this thesis we use three distinct priority levels.

The priority mechanism makes sure that real-time applications, such as e.g. low latency audio streams, are preferred over non real-time applications, such as e.g. bulk downloads.

### 4.3.2. Latency

Another concern of predictability is latency. Especially for real-time systems, latency is often a critical factor to consider. Therefore, the interface should minimize the end to end latency of submitting and receiving operations to and from the network stack.

This can for example be achieved by using lock-free data structures instead of locking, as locking may make latencies less predictable.

### 4.3.3. Fault tolerance

Another aspect of predictability is fault tolerance. Fault tolerance means that a crash, timeout or misbehavior of the application that uses the interface will have no influence on functionality of the interface for other applications using it.

When an application that uses the interface crashes, the interface must handle this accordingly. This means the crashed application should not for example cause the interface to hang.

Misbehavior of the application should be handled gracefully by the interface. For example, an application that is stuck in a deadlock and can therefore not interact with the interface for a certain time, should not cause an issue in the interface or network stack.

For network stacks running in kernel space, fault tolerance is less of a problem than it is for user space stacks, because the kernel usually offers a robust system call interface on which the network stack can rely on.

### 4.3.4. Interference avoidance

Avoiding any interference between the applications that use the interface is another aspect of predictability. Interference between applications generally makes the latency prediction of flows in advance more complicated.

Furthermore, interference between applications can lead to undesired side effects, may it be in terms of performance or in terms of security.

Interference can for example be data structures that are shared between applications or global locks.

### 4.3.5. Application multiplexing

The interface has to multiplex applications and make sure that the scheduling requirements mentioned in Section 4.3.1 are respected irrespective of which application a socket belongs to. Application multiplexing describes the process of merging operations from multiple applications, which in turn have multiple sockets of different priorities, into one operation pipeline.

## 4.4. Functionality

The interface should offer all TCP operations possible. These consist of:

- `bind`: Bind a socket to an address
- `listen`: Put a socket into listening mode
- `accept`: Accept new connections on a socket
- `connect`: Connect to another socket
- `write`: Write data to a socket
- `recv`: Receive data from a socket
- `close`: Close a socket

This means the stack is required to have the same TCP functionality as usual network stacks, such as the Linux network stack for example. The implementation of socket options is not necessary, but the API should be designed so that such options can be added in the future.

The interface should be extendable with only small development effort, offering the possibility to integrate UDP sockets, raw sockets or other protocols that lwIP offers.

## 4.5. Security

The interface should offer certain security guarantees. For example, applications using the interface should not be able to read data that other applications have sent or received through the interface.

Moreover, it should not be possible for a malicious application to use a DoS attack to make the interface unusable for other applications.

Both of these goals go hand in hand with the requirements related to interference avoidance.

## 4.6. Further non-functional requirements

Further non-functional requirements that the interface should fulfill are:

- Usage of state-of-the-art synchronization methods, which minimize the overhead for synchronization

- Usage of state-of-the-art communication methods, which minimize the overhead for communication
- Minimize the amount of memory copies of data that is sent or received through the interface
- Ability to port the interface to other operating systems than Linux, such as e.g. microkernels

# 5. Design and Implementation

This chapter presents the design and implementation decisions that were made when developing *pifus*.

Section 5.1 starts with an overview about the high level architecture and then it goes into more details concerning single aspects of *pifus*.

Section 5.2 discusses alternative approaches, while Section 5.3 discusses design decisions regarding the portability of the interface.

## 5.1. Architecture

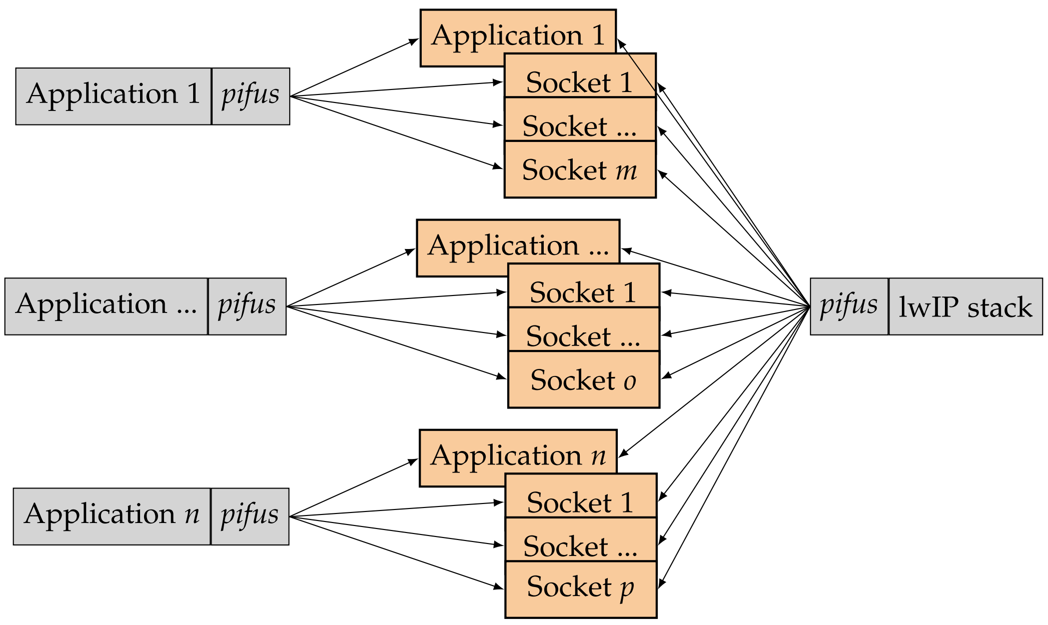

Figure 5.1.: Overview of applications interacting with the lwIP stack using *pifus*.

As seen in Figure 5.1, *pifus* makes the lwIP stack accessible for any application linking against the *pifus* client API. The design and implementation of the client API itself is described in 5.1.4. The client API uses inter-process communication to communicate with the *pifus* backend.

For each application, one shared memory area is created. Additionally, for each socket that the application uses, another shared memory region is used. In Figure 5.1, processes are depicted in gray, whereas shared memory regions are drawn in orange. The details of inter-process communication are outlined in Section 5.1.1 for shared memory and in Section 5.1.2 for synchronization.

The *pifus* backend resides in the same process as the lwIP stack. Features of the backend are for example connecting to the appropriate lwIP API, implementing QoS guarantees or the discovery of new *pifus* applications. See Section 5.1.5 for a detailed explanation of the features.

Dynamic memory allocation is inherently important for interaction with the network (e.g. for sending and receiving data). Therefore, Section 5.1.3 describes how *pifus* tackled this aspect. Furthermore, the main data structures used in *pifus* are also explained in Section 5.1.3.

### 5.1.1. Shared memory

*pifus* uses shared memory as defined in the POSIX standard [27], [28]. Shared memory was chosen over other communication techniques simply because it offers high performance due to the direct memory access. This fulfills the requirement of the analysis to reduce the overhead for communication which was set in Section 4.6.

On the other hand, a downside of shared memory is that one has to ensure that processes that are simultaneously accessing or modifying data inside the memory do this on a synchronized basis to avoid race conditions.

Where necessary, *pifus* achieves this by using lock-free data structures inside the shared memory regions. This has the advantage that for accessing the shared memory regions, no expensive locking mechanisms such as mutual exclusions are needed.

There are two types of shared memory regions in *pifus*:

- The **application region**, which holds information to detect the socket regions and serves as space for dynamic memory allocation.
- The **socket region**, which holds the complete state of a single socket.

#### Application region

One application region is created for each application that uses *pifus*. Its main purpose is to store data that needs to be shared between the application and the *pifus* backend. This is for example the case when the application wants to `write` to a socket. Other applications do not have access to this region, making the data not accessible for applications other than the backend and the application itself, fulfilling the goal of data confidentiality set in Section 4.5. The memory allocation algorithm used inside the region is explained in Section 5.1.3.

Other than that, the application region only holds an `uint32_t` that holds the number of sockets that this application has opened. This is used as a futex (a lightweight mutex, see Section 5.1.2) to signal the backend that a new socket was created. It is not decremented when a socket is closed.

### Socket region

The socket region is created on a per-socket basis. Most importantly, it stores information about the state and the type of socket, including:

- The type. (currently only TCP)
- The priority. (see Section 5.1.5)
- An unique identifier tuple of two `uint32_t`, similar to a file descriptor.
- The **submission queue**, containing operations queued from the client side.
- The **completion queue**, containing finished operations queued from the backend side.

Both the **submission queue** and the **completion queue** are implemented as ring buffers, see Section 5.1.3 for details. The queues are used to implement an asynchronous I/O API as it was required in Section 4.1.

An overview about the shared memory regions and the interaction between them and *pifus* is depicted in Figure 5.2.

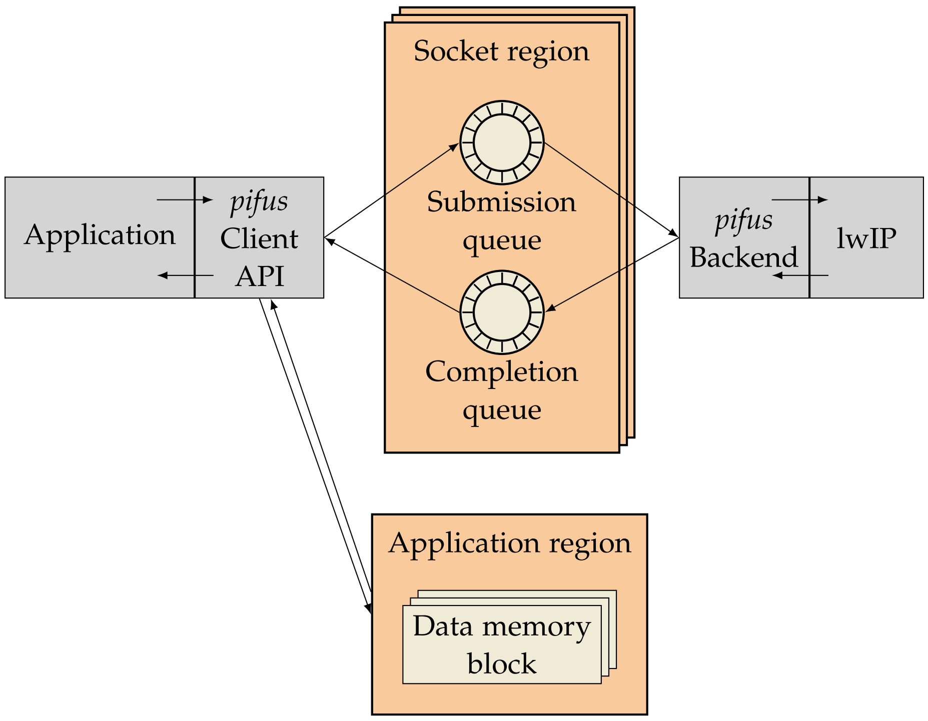

Figure 5.2.: Overview of communication between *pifus* entities. Processes are depicted in gray, while shared memory regions are drawn in orange.

The data flow of the queues is depicted in Figure 5.3.

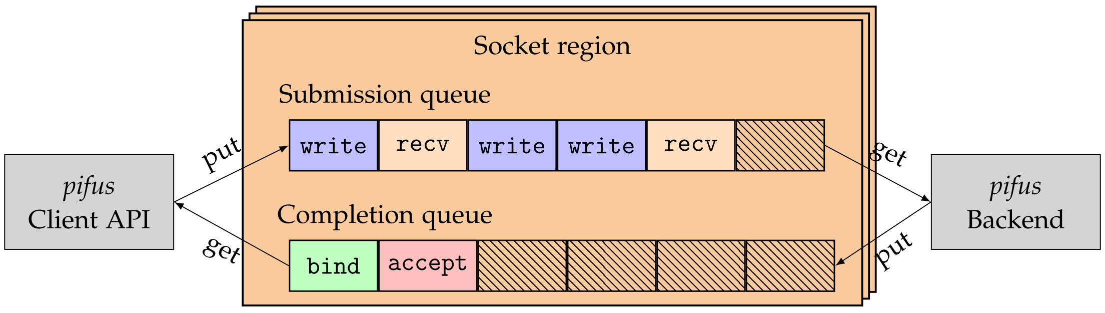

Figure 5.3.: Data flow between the *pifus* client and the *pifus* backend using the submission and completion queue.

Furthermore, the socket region also stores information that is only used in the backend to have all socket information available in a single place. This includes:

- queues to keep track of asynchronous operations, such as `write` or `receive`

- a receive buffer, which stores incoming data even if no `recv` operation has been enqueued by the client
- a pointer to the lwIP `pcb` [12, Section 10.2] corresponding to the socket
- the number of operations enqueued by the client
- the number of operations dequeued by the backend

For further details about the usage and intent of these variables see Section 5.1.5.

### 5.1.2. Synchronization

A fast userspace mutual exclusion (futex) is used to efficiently signal interrupts between threads and processes. Compared to standard mutexes, futexes implement uncontended lock operations in user space. Therefore, they require less context switches and have (on average) lower overheads than other synchronization mechanisms. They were chosen to fulfill the requirement of minimizing the time for IPC synchronization set in the analysis in Section 4.6.

The relevant system calls when working with futexes are:

- `futex_wait` [29], which waits for a specific futex to be signaled
- `futex_wake` [29], which signals a futex

`futex_wait` takes an expected value of the futex variable. If the expected value does not match the actual value of the futex, the call immediately returns `EAGAIN`. Else, the call will return when either the given timeout (if any) has passed or another thread changes the futex variable and calls `futex_wake`.

A futex itself is an unsigned integer variable, usually with a size of 32 bits.

With this original implementation inside the Linux kernel, it was only possible to wait for a single futex to be woken. The Linux kernel version `5.16` introduced the `futex2` API, which most importantly adds the system call `futex_waitv` for waiting on multiple futexes. It supports to wait for up to 128 futexes. [30]

*pifus* makes extensive use of the `futex2` API and futexes in general. For instance, they are used to signal that a process has added a new item to a queue. For details on where exactly futexes are used in *pifus*, see Section 5.1.3.

### 5.1.3. Data structures

The following subsections elaborate on the data structures used inside *pifus*.

### Ring buffer

The ring buffer implemented in *pifus* is a FIFO queue of fixed size, as it works on an underlying C array. It is inspired by the open-source ring buffer published by QuantumLeaps [31]. The size of the queue can be defined at compile time.

Due to the use of C macros, the ring buffer itself supports holding elements of any predefined type.

```c
#include <stdint.h>
struct ring_buffer {
    uint16_t end;
    uint16_t head;
    uint16_t tail;
};
```

Listing 5.1: Control structure of the ring buffer.

The control structure of the ring buffer is depicted in Listing 5.1.

The *end* variable stores the length of the array. This means that $end - 1$ is the last index in the underlying array that is used by the ring buffer. While the *head* variable is always pointing to the last element that has been added to the queue, the *tail* variable has the invariant of being set to the element that is the next one to be dequeued.

The underlying array of the ring buffer is not contained in the control structure. Instead, it is supplied in every call to the ring buffer. This is necessary because the ring buffer may lay in shared memory, and a pointer stored inside the control structure would then only be valid for one process using the ring buffer.

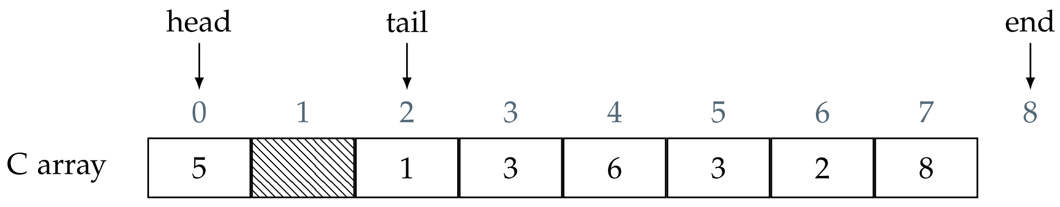

Figure 5.4.: Ring buffer filled with example integer data. The underlying array has a length of eight.

A simple example instantiation of the ring buffer is shown in Figure 5.4. This figure also shows a scenario where *overstepping* occurred. *Overstepping* happened when the `head` is less than the `tail`, i.e. when the current data stored in the ring buffer is distributed across the array bounds. While *overstepping* is not possible with only inserting data, it is possible when having insertions and deletions from the ring buffer in a certain order.

The ring buffer interface offers the following API:

- `create`: initializes the ring buffer control structures
- `get`: dequeues the next element in a FIFO order and stores it at a given pointer
- `peek`: returns a pointer to the next element without dequeuing it
- `peek_index`: returns a pointer to the element at the given index without dequeuing it
- `erase_first`: dequeues the next element in a FIFO order without storing it
- `put`: inserts a given element into the ring buffer
- `find`: returns a pointer to the first item satisfying a given condition
- `is_full`: returns whether the ring buffer is full or not

There are two basic operations that are essential to understand the whole concept of the ring buffer, namely `put` and `get`.

The pseudocode of the operations is given by Algorithm 1 and respectively Algorithm 2.

**Algorithm 1** Algorithm for inserting an element into the ring buffer

```
$ring\_buffer$: The ring buffer control structure.
$data$: The underlying array.
$element$: The element to insert.
procedure PUT($ring\_buffer, data, element$)
    $head \leftarrow ring\_buffer.head + 1$
    if $head = ring\_buffer.end$ then
        $head \leftarrow 0$                                ▷ Account for overstepping
    end if
    if $head \neq ring\_buffer.tail$ then             ▷ Check if ring buffer is not full
        $data[ring\_buffer.head] \leftarrow element$
        $ring\_buffer.head \leftarrow head$
        return $true$
    end if
    return $false$
end procedure
```

**Algorithm 2** Algorithm for dequeing an element from the ring buffer

```
$ring\_buffer$: The ring buffer control structure.
$data$: The underlying array.
procedure GET($ring\_buffer, data$)
    $tail \leftarrow ring\_buffer.tail$
    if $ring\_buffer.head \neq tail$ then                ▷ Check if ring buffer is not empty
        $element \leftarrow data[tail]$
        $tail \leftarrow tail + 1$
        if $tail = ring\_buffer.end$ then
            $tail \leftarrow 0$                         ▷ Account for overstepping
        end if
        $ring\_buffer.tail \leftarrow tail$
        return $element$
    end if
    return $\emptyset$
end procedure
```

This implementation allows the ring buffer to be lock-free under the following conditions:

1. There is a single producer and a single consumer. This means at most one thread is adding items to the buffer, while at most another one is dequeuing data from the buffer.
2. The state variables of the ring buffer can be written atomically by one machine instruction. This is the case when the size of the variables inside the control structure does not exceed the CPU architecture's word size and the struct is not misaligned.

In *pifus*, the ring buffer is solely used so that no locking is needed to minimize the latency as mentioned in the analysis in Section 4.3.2.

Moreover, all ring buffers used in *pifus* are used in combination with a futex for indicating how many elements were added to the ring buffer. This way the consumer of the ring buffer can get notified through futexes when there is a new entry in the buffer and can then dequeue it.

#### Byte buffer

The byte buffer is similar to the ring buffer that was presented in Section 5.1.3. Different to the ring buffer already presented, the goal is to be able to efficiently push and pop a dynamic amount of bytes, instead of just one element, to and from the buffer.

The implementation uses the same control structures as in Listing 5.1. The main difference in the implementation, not in the control structure, is that the memory copying into the buffer is not done for each element (in this case, a byte) that is added to the buffer. Instead, the byte buffer tries to group the `memcpy` calls to achieve the goal of minimizing memory copies as defined in Section 4.6. When no overstepping occurs, only one `memcpy` call is needed. With overstepping, at most two `memcpy` calls are needed.

### Linked List

*pifus* additionally makes use of a doubly linked list. As seen in Section 5.1.5, the linked list in *pifus* is only used in the priority threads. The implementation of the list was obtained from a collection of open-source C data structures [32].

The list itself is a simple doubly linked list, offering an API that is able to push and pop elements to and from the front and the back of the list in constant time.

For storing the data, memory is allocated dynamically on the heap using `malloc`.

### Dynamic memory allocation

As described in Section 5.1.1, *pifus* manages dynamically allocated memory inside the **application region**. The backend as well as the client API allocate, access and may free data inside this shared memory region.

The *pifus* memory management API consists of `allocate` and `free` methods. The API is similar to the `malloc` and `free` functions contained in the C standard library defined in `stdlib.h`.

The control structures of the memory management are solely based on offsets inside the shared memory region. This is needed as one is not able to store pointers in the control structures in shared memory, as different processes that access the area may have different virtual addresses pointing to the same physical address. Accessing a virtual address from another process would lead to an invalid memory access.

```c
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>

typedef int64_t block_offset_t;
struct pifus_memory_block {
    /* Indicates whether this block is free or not. */
    bool free;
    /* The size in bytes of this block. */
    size_t size;
    /* Block offset (absolute) of the previous block.
    * -1 if there is no previous block. */
    block_offset_t prev_block_offset;
};
```

Listing 5.2: A *pifus* memory block

The `allocate` method allocates a coherent region of memory, a so-called *block*, of a specific size inside the shared memory region. The structure of a *block* can be seen in Listing 5.2. The blocks are essentially elements in a doubly linked list, where `size` implicitly yields the start of the next block by doing simple arithmetics and `prev_block_offset` explicitly gives the start of the previous block. Using these two variables, one can iterate through the whole memory layout.

`allocate` uses a *first fit* algorithm. This means the first free block able to satisfy the allocation is used. In case there is no suitable block or the end of the chain of blocks is reached and assuming the space of the shared memory region is not exhausted, a new block is created at the end. The search for a free block always begins at the start of the shared memory region.

`free` removes a block from the memory region. Removing a block can either mean that:

1. the block's `free` variable will be set to `true`, or
2. the block will be merged with neighbor blocks that are unused, or
3. the block will be deleted and its area will be `memset`ted with zeros, because it was the last block in the chain

Merging neighbor blocks that are free is useful because the merged block has a larger size and can then potentially satisfy a larger allocation. Due to merging, a free block never has a neighbor that is also free.

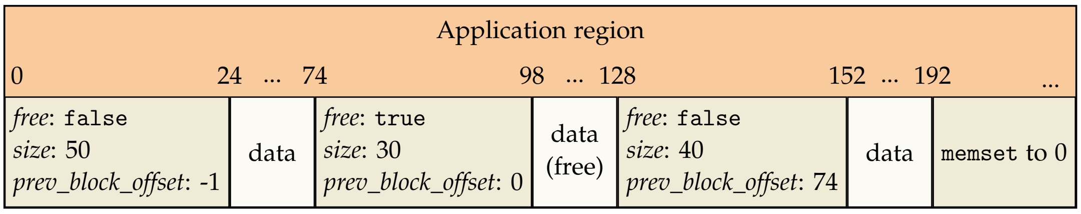

Figure 5.5.: Example memory layout using the described algorithm. The numbers at the top indicate the byte offset.
*Note:* The **application region** contains another variable prior to the allocated memory which is omitted in this figure for simplicity.

An example instantiation of the algorithm is depicted in Figure 5.5. Note that `sizeof(struct pifus_memory_block)` equals 24 bytes in this example. In this example, there are three allocated blocks, of which one was `freed` and therefore does not contain any data. Further blocks could be `allocated` by creating blocks after the last block's data at byte *192*.

In general, `alloc` and `free` can not be implemented without using locks. Nonetheless, *pifus* is designed so that the `alloc` and `free` methods are solely used by the client API. In this way, no locking is needed under the assumption that a socket is always used exclusively by a single client thread and not shared among client threads.

For more details on how the algorithm is used on the client side API, see Section 5.1.4. As the backend only reads existing blocks and a concurrent `free` on the client side of that block is ruled out, no locking is needed for coordinating the backend and client access.

### 5.1.4. Client API

The *pifus* client API exists in form of a library that can be linked. The library must be linked to communicate with the *pifus* backend and therefore access the network stack.

#### Structure

The API is inspired by the BSD socket API and offers similar functionality. The main difference to many existing socket APIs is that it is based on a completion I/O model instead of a readiness I/O model, as it was required by Section 4.1.

This implies that calls to the API return immediately after the requested operation has been enqueued. In *Callback mode*, the application is notified upon completion of the requested operation, whereas in *Poll mode*, the application can do a busy wait on the result or try to dequeue it at any given time, but is not notified.

Possible operations on a *pifus* socket are:

- `pifus_socket`
- `pifus_socket_bind`
- `pifus_socket_connect`
- `pifus_socket_write`
- `pifus_socket_recv`
- `pifus_socket_listen`
- `pifus_socket_accept`
- `pifus_socket_close`

Their semantics is identical to the corresponding BSD sockets semantics, so that porting existing applications is easy and developers get quickly used to the API. This helps to fulfill the goal of simplicity set in the analysis chapter in Section 4.1. For detailed method signatures and more information, please refer to the header files `pifus.h` and `pifus_socket.h`.

**Poll mode**

In poll mode, the client API only runs on a single thread. The user of the API is responsible for dequeuing finished operations, as *pifus* has no way to notify the application.

The API offers functionality to block the main thread and wait for a new result in a specific socket to be available. When no waiting is desired, the user of the API can also manually check for a new result for a specific socket at any given time.

The poll mode generally has the advantage that no extra thread is needed, but comes with the downside of the developer having to deal with new results arriving in an asynchronous fashion.

**Callback mode**

Contrary to the poll mode, the callback mode starts an additional thread. This thread is responsible for invoking a callback specified by the user when there is a new result in any of the created sockets. This is achieved by waiting on the futexes of all sockets' completion queues using `futex_waitv`. If one of these futexes is woken, the callback is called for that specific socket. The usage of `futex_waitv` also implies that the callback mode has the limitation to only support notifications for up to 128 sockets.

Generally, the callback mode has the advantage that the user of the API does not have to manage checking the completion queues of all the sockets, but can rely on *pifus* to send a notification.

### Internal implementation

Internally, the *pifus* client API uses the shared memory regions for communication with the backend (see Section 5.1.1).

When initializing, the API automatically creates a shared memory region for the application. This application region is named after a certain pattern, namely `appX`, where the `X` is an ascending integer incrementing for each started *pifus* application. The application creating the application region therefore looks for other shared memory regions already present on the system, and then sets `X` to be the lowest non-existing shared memory region. The prototype of *pifus* does not support concurrent application startups, but this could be fixed with e.g. a global lock in the file system.

Upon the creation of a socket via `pifus_socket`, a new shared memory region is created for that socket. The region is named after the pattern `appX-socketY`, where `X` indicates the application index and `Y` indicates the socket index. Furthermore, the futex, contained in the application region, that represents the amount of sockets that this application has opened, is incremented and woken. This way, the backend is notified when a new socket is opened by an application. For further details, refer to Section 5.1.5. The increment of the futex is not guarded by locks, making the `pifus_socket` call not thread-safe. *pifus* makes the assumption that sockets are only created by a single application thread. The user may overcome this assumption by guarding the `pifus_socket` calls with locks to coordinate access from multiple threads.

For a graphical illustration of the shared memory regions, refer to Figure 5.1.

When executing any operation on a socket using the client API, *pifus* inserts an *operation struct* into the submission queue. The operations available are listed in Section 5.1.4. The *operation struct* contains specific information about the operation that the user wants to execute, e.g. a `recv` operation would contain an offset to the memory block that should contain the received data after the operation has been successfully executed.

When the user wants to execute an operation, but the submission queue is full, the call returns `false`. After the operation has been queued, the backend processes the operation, and when finished, inserts the operation result into the completion queue of a particular socket. Figure 5.3 shows the interaction between the client API and the backend using the submission and completion queue. Note that the *operation result* *struct* contains a code that indicates if the operation was successful, and may contain further operation specific data. For example, the *operation result struct* for a `write` operation contains the offset of the memory block of the data that was written. This way the client API can `free` the memory related to that certain `write`.

As described in Section 5.1.3, when something is added to any of the queues, a futex is woken. This ensures that both the backend and the client potentially get to know that the other side has added new entries to a certain queue. See Figure 5.6 on how futexes are used together with the queues.

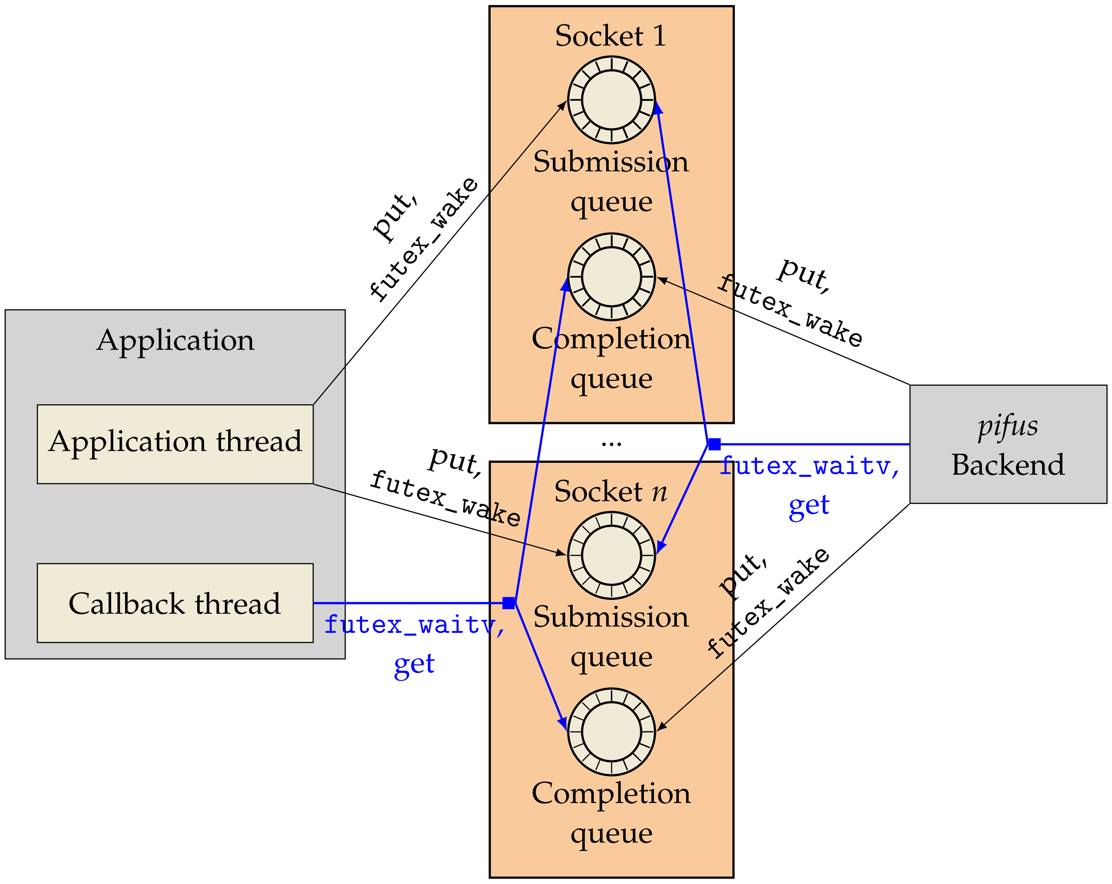

Figure 5.6.: Usage of futexes with the *pifus* client API in callback mode.

The procedure explained in previous paragraphs indicates that the backend is designed so that any application that does not interact with the backend anymore (e.g. due to a deadlock or crash of the application) does not hinder the processing of the backend. This is because the only communication between the two entities happens over the queues, and if the application does not interact with the queues anymore, the backend also does not have to. This supports the goal of fault tolerance set in Section 4.3.3.

### Memory management

The client API uses the memory allocation algorithm presented in Section 5.1.3. The API tries to do as much memory management itself rather than letting the user do it.

The two operations of the API that require memory allocations are `write` and `recv`.

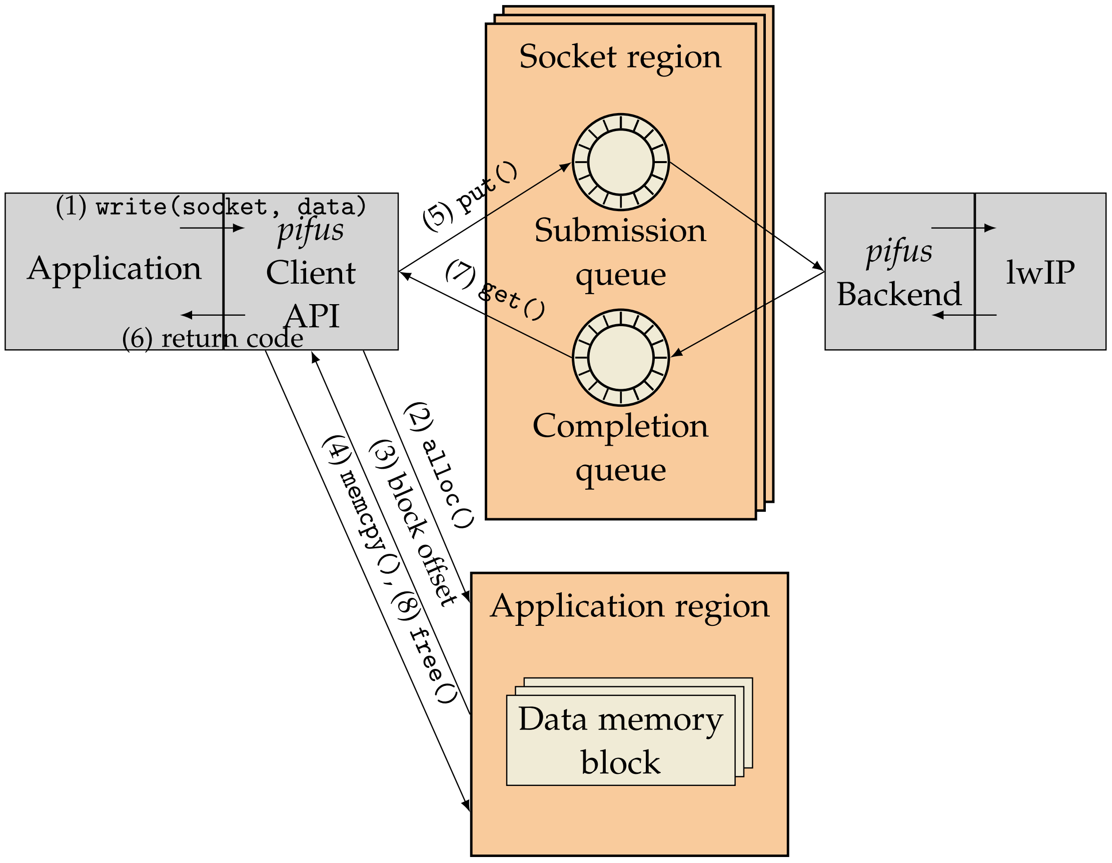

Figure 5.7.: Sequence of events regarding memory management when executing a `write`.

See Figure 5.7 for `write`. The numbers in the next paragraph refer to the figure. When executing a `write` (1), internally a memory block is allocated (2, 3) with the size of data that the user wants to write. The data is then copied from the data pointer supplied by the user into the newly allocated memory block. (4) The offset of this memory block is then passed to the backend inside the submission queue entry. (5) Afterwards, the client API call returns (6) and indicates if the operation has successfully been queued. After the `write` operation itself succeeds (that is when the receiver of the data has `ACK`ed it) and there is a corresponding entry in the completion queue received by the client API (7), the memory block is freed again (8).

Similarly, for `recv`, a memory block is allocated with the size that the user wants to receive data with. The offset is also passed to the backend and the memory block related to this offset is then filled with data as soon as it arrives at the socket. In contrary to `write`, the memory block has to be freed by the user itself after the operation finished. This is because the user can decide to use the data as long as she or he wishes to and therefore has responsibility for the data.

For both of these operations, exactly one copy is needed. *pifus* tried to minimize the amount of copies as much as possible due to the requirement set in Section 4.6.

### Example usage

An example usage of the *pifus* client API in callback mode is shown in Listing 5.3. In this example, a TCP socket with high priority is opened and then used to connect to another machine. After connecting, it will endlessly write a certain string on the socket.

```c
#define _GNU_SOURCE
/* standard includes */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
/* pifus includes */
#include "pifus.h"
#include "pifus_ip.h"
#include "pifus_socket.h"

void callback_func(struct pifus_socket *socket,
                   enum pifus_operation_code op_code) {
    // Check if the operation that has finished is a write.
    if (op_code == TCP_WRITE) {
        // Pop the result from the completion queue.
        struct pifus_operation_result result;
        pifus_socket_pop_result(socket, &result);

        if (result.result_code == PIFUS_OK) {
            printf("Written another line!\n");
        } else {
            printf("Something went wrong, the write did not succeed.\n");
        }
    }
}

int main(int argc, char *argv[]) {
    char *reader_ip = "192.168.2.1";
    uint16_t port = 13337;

    // Use the callback mode.
    pifus_initialize(&callback_func);

    // Open a new TCP socket with a high priority.
    struct pifus_socket *socket = pifus_socket(PROTOCOL_TCP,
                                               PRIORITY_HIGH);

    // Bind the new socket to a valid IPv4 address on a free port.
    pifus_socket_bind(socket, PIFUS_IPV4_ADDR, 0);

    struct pifus_operation_result operation_result = {};
    /* Wait for the bind operation to succeed and write its result in the
    operation_result variable. This only works if we do not dequeue the
    result in the callback above.
    We do not want to use the callback here because we want to block
    until the socket is bound. */
    pifus_socket_wait(socket, &operation_result);

    // Read the IP from a char* into an pifus_ip_addr struct.
    struct pifus_ip_addr remote_addr;
    ip_addr_from_string(reader_ip, &remote_addr);

    // Connect the socket to the address and wait until we are connected.
    pifus_socket_connect(socket, remote_addr, port);
    pifus_socket_wait(socket, &operation_result);

    while (true) {
        int sent = 0;
        while (sent < 10) {
            char *loop_data;
            if (!asprintf(&loop_data, "Predictable interface for a user
                space IP stack!#%i", sent)) {
                // error handling
            }

            // Write the string that is stored above to the socket.
            if (pifus_socket_write(socket, loop_data,
                strlen(loop_data))) {
                sent++;
            }

            free(loop_data);
        }
    }

    // Close pifus gracefully.
    pifus_exit();

    return 0;
}
```

Listing 5.3: Example *pifus* Client API usage.

### 5.1.5. Backend

The *pifus* backend is the main component of *pifus*. It integrates the lwIP stack with the rest of *pifus*, while also ensuring predictability using a QoS mechanism based on priorities. It is a standalone executable, which in particular links against the lwIP stack.

#### Design

The backend starts multiple threads that are responsible for different tasks. There are three different types of threads:

- The *stack thread*. This thread interacts with the lwIP API and is the main thread of the backend.
- The *discovery thread*. It is responsible for detecting newly started *pifus* clients and newly created sockets in these clients.

- The *priority threads*. A thread is started for each priority. Each thread is responsible for handling new operations submitted by sockets of a certain priority. For details regarding priorities, see Section 5.1.5.

In total, the *pifus* backend therefore starts

$$n_{threads} = 2 + n_{priorities}$$

threads, where $n_{priorites}$ is the number of different priority levels. In Figure 5.8, the thread structure is depicted for three different priorities.

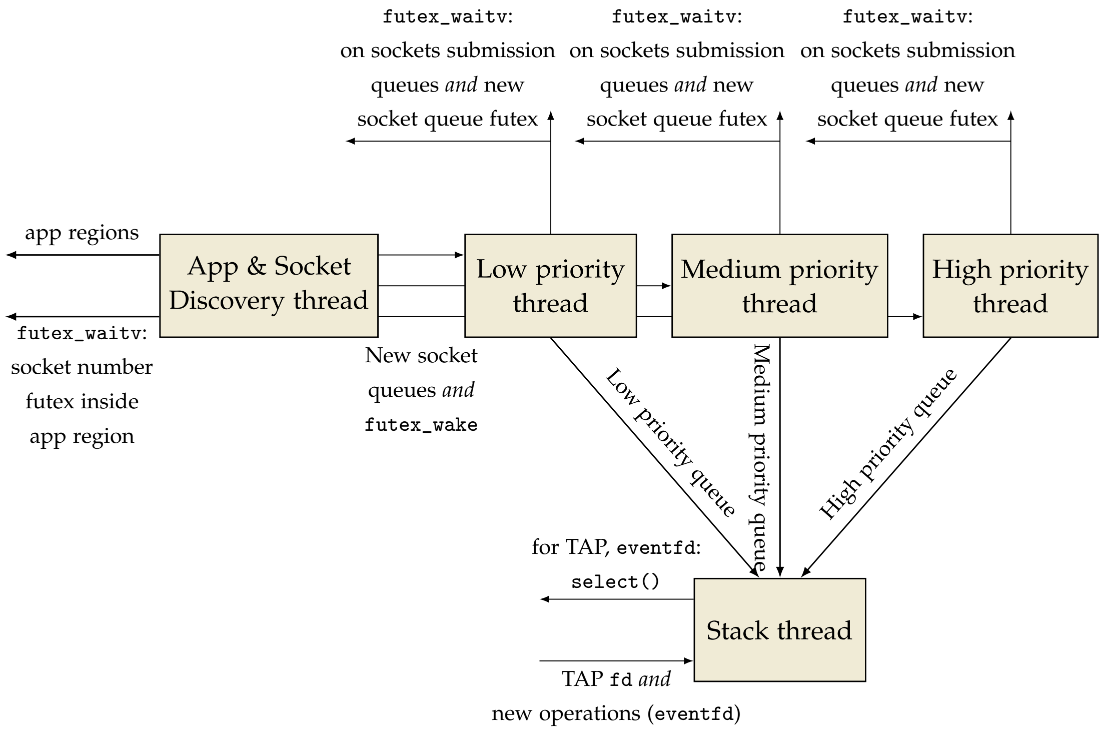
futex_waitv:
on sockets submission queues *and* new socket queue futex
futex_waitv:
on sockets submission queues *and* new socket queue futex
futex_waitv:
on sockets submission queues *and* new socket queue futex
app regions
App & Socket Discovery thread
Low priority thread
Medium priority thread
High priority thread
futex_waitv:
socket number futex inside app region
New socket queues *and* futex_wake
Low priority queue
Medium priority queue
High priority queue
for TAP, eventfd:
select()
Stack thread
TAP fd *and*
new operations (eventfd)

Figure 5.8.: Threads of the *pifus* backend and their communication with each other and with other *pifus* entities.

It can be seen that the communication between the threads relies on queues and futexes. This is the same pattern as in the communication between the client and the backend. The ring buffer presented in Section 5.1.3 is used as an implementation of these queues.

When a new socket is opened by an application on the client side, it increments the socket counter in the application region that the discovery thread waits upon. The discovery thread then wakes up and puts the socket into the `new_socket_queue` of the priority that the socket belongs to and wakes the futex related to that queue. The corresponding thread responsible for the priority then adds the socket to a linked list as described in Section 5.1.3 and uses `futex_waitv` to wait on the submission queue futexes of all sockets. When there is an operation queued from the client side in the submission queue, the priority thread wakes up and puts this operation into the corresponding `priority_queue`. The stack thread can then execute the operations by popping them from the priority queues and then calling the lwIP API. Note that a priority thread supports at maximum 127 concurrent sockets due to the usage of `futex_waitv`, which supports a wait on up to 128 futexes. Another wait on futex is used to wait on the futex related to the `new_socket_queue`.

The stack thread runs the lwIP logic and uses `select` [1] to poll the driver's interface. This `select` would only return after the timeout has passed or there is something new on the driver's interface. Of course, there is also the possibility that in the meantime new operations are queued from the *pifus* client side. Therefore, to not delay the processing of new operations, the call to `select` also includes an `eventfd` [33]. This `eventfd` is written to by the priority threads in case a new operation has been received. The `eventfd` is only needed when using the TAP interface. It is not needed with interrupt based drivers.

### lwIP API

*pifus* uses the raw API [18] of lwIP. Furthermore, lwIP is set to use the mainloop mode (`NO_SYS` [34]). The mainloop mode allows full control over lwIP threading, this means that the whole lwIP context runs on a single thread completely managed by *pifus*. Access to the raw API is not thread-safe. This implies that all interaction with lwIP has to happen on a single thread, namely the stack thread.

The stack thread is therefore responsible for:

- Polling the Ethernet driver.
- Periodically invoking lwIP to check TCP timers and trigger callbacks.
- Dequeuing incoming operations from the priority queues of the different priority threads and passing them to lwIP.

Since all the network stack logic is running on this thread, the amount of *pifus* operations performed in the stack thread is optimized and bounded to allow sufficient time for the network handling. This is achieved by limiting the amount of operations that can be popped during an iteration of the thread loop. Furthermore, *pifus* limits the amount of parallel operations on a per-socket basis.

The *pifus* prototype implements full TCP support to fulfill the requirements set in Section 4.4, but omits UDP support for the time being. As UDP has less complexity, it can be seen as a simplification of TCP and would be trivial to implement into *pifus*.

The remainder of this subsection is about how *pifus* implements each of the supported TCP operations using the lwIP raw API.

For each *pifus* socket that is created, a TCP `pcb` is created using the `tcp_new` function of the lwIP raw API. A `pcb` is the abstraction of a connection in lwIP. Any TCP functions can be called on that `pcb` using lwIP.

*pifus* disables *Nagle's algorithm* for each socket. This is done because the algorithm may maximize throughput, but especially for small packets may increase latency. Therefore, *Nagle's algorithm* is disabled per default. While currently not implemented, it would be possible to enable the algorithm for certain sockets, e.g. through a flag in the *pifus* client API.

When the stack thread dequeues a `bind` operation, the `tcp_bind` function [35] is invoked. The `tcp_bind` function in the lwIP raw API is synchronous, this means that the result of the operation is immediately returned by lwIP. This does not hold true for all other lwIP operations as we are going to see later on. When executing a synchronous lwIP operation, the stack can immediately insert the result of the operation into the completion queue of the corresponding socket. Other synchronous operations used are `tcp_listen` and `tcp_close` [35].

In contrast, the `tcp_connect` method [35] of the lwIP raw API is an asynchronous function. This means, the method immediately returns after executing, but one only knows the result of the `connect` operation after the corresponding callback has been called. In this case, the stack thread inserts the result of the operation into the completion queue after the callback has been called. Other asynchronous operations that are used are `tcp_accept`, `tcp_write` and `tcp_recv`. [35]

For some asynchronous functions, a queue of pending operations has to be created in order to know which operation is the next one to fulfill when the callback is called. In the case of the `write` operation, this queue is called the `write_queue`.

For example, let's assume the following order of `write` calls of a *pifus* client, which the stack thread then has to pass to lwIP:

1. `write` { *data_block_offset = 0, size = 100* }
2. `write` { *data_block_offset = 130, size = 10* }
3. `write` { *data_block_offset = 140, size = 5* }
4. `write` { *data_block_offset = 1337, size = 15* }

The stack thread would call `tcp_write` for each of the operations, which would return immediately due to it being asynchronous. When the callback is invoked, *pifus* has to remember which `write` operations it previously queued, so that the corresponding operation result can be put into the completion queue with the right *data_block_offset*. The *data_block_offset* refers to the memory block of the data that has been written, while *size* refers to the size of the memory block. The callback that is invoked contains the length of the data that has been written. Now assume, after submitting the four operations from above, the callback is invoked with a written data length of `110` bytes. Then, *pifus* should put operations 1. and 2. into the completion queue, so that the data can be freed by the client API. This is achieved by popping as many operations as needed from the above-mentioned queue so that the length passed in the callback is satisfied. In our particular case *pifus* would pop the queue two times so that the remaining length would yield zero:

$$remaining\_length = acked\_length - write_1\_length - write_2\_length = 110 - 100 - 10 = 0$$

It would be fatal to put a `write` operation into the completion queue that has not yet been `ACKed` by the counterpart, because after the client API dequeues this operation, the corresponding memory is freed. If the data has not been `ACKed` yet, it would be lost at this time and lwIP would not be able to transmit it, because it takes the data directly from the shared memory without copying it.

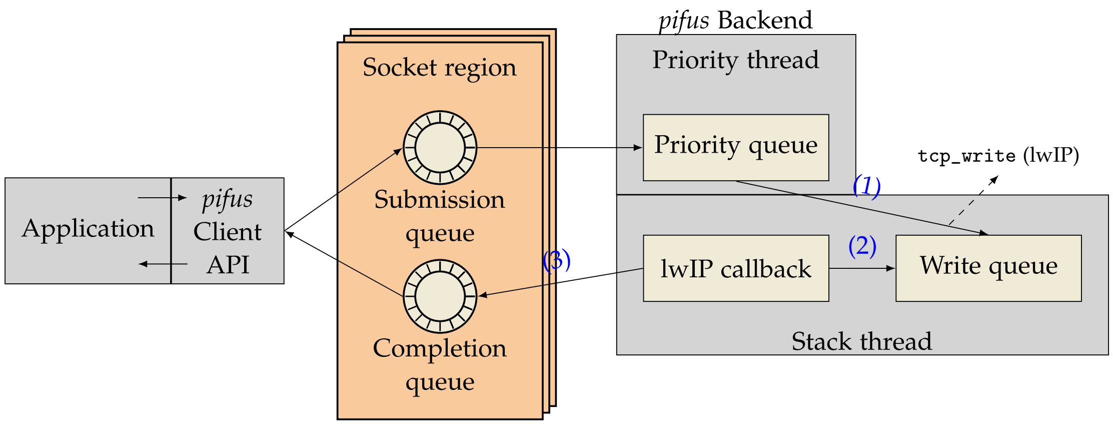

Figure 5.9.: Flow of a `write` operation on a TCP socket.

Figure 5.9 is a depiction of the data flow when a `write` operation is executed. The numbers in the paragraph below correspond to the numbers depicted in the figure.

Therefore, every time a `tcp_write` is called, the operation leading to that call is enqueued in the `write_queue` (1). When the callback is invoked, the above described algorithm is executed (2) and the `write` operations that are finished are put into the completion queue (3).

The queue is implemented using the ring buffer that was described in Section 5.1.3.

Further, note that a call to `tcp_write` of the lwIP API may directly fail, e.g. when the send buffer of the lwIP stack has run full. *pifus* then puts an operation result with an error code into the completion queue. In this case, the `write_queue` is bypassed and nothing is inserted into it. Furthermore, *pifus* detects when the send buffer of lwIP is full and tries not to execute any further `write` operations until the send buffer has emptied.

Another asynchronous operation that makes use of such a queue is the `recv` operation. The queue for pending `recv` operations is called `recv_queue`. The process of how the queue is managed is similar to the `write_queue` that we have seen before. Additionally, `recv` implements a buffer for incoming data to the socket when no `recv` call has been issued by the client side. This is similar to the buffers in network stacks implemented in e.g. Linux. For buffering incoming data, the byte buffer described in Section 5.1.3 is used.

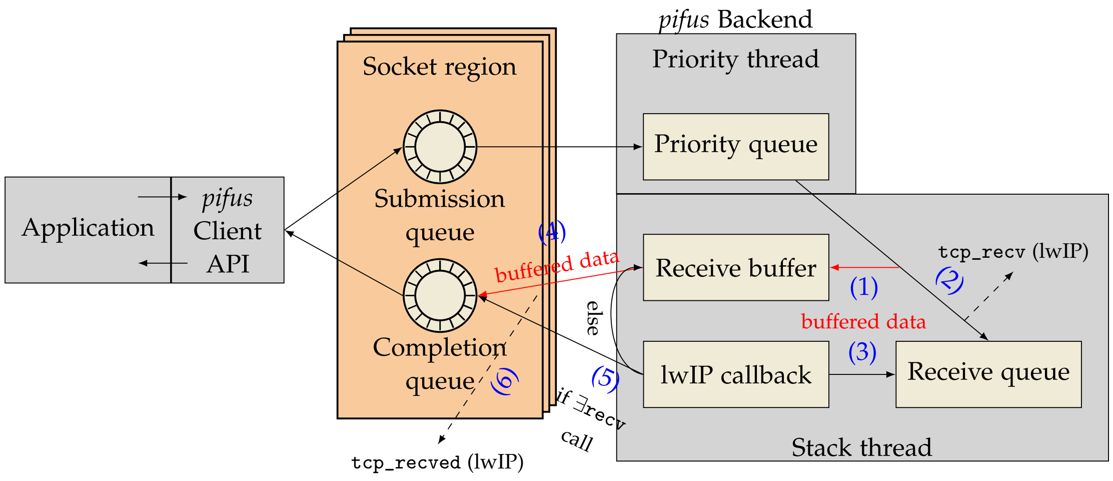

Figure 5.10.: Flow of a `recv` operation on a TCP socket.

Figure 5.10 shows the data flow for the `recv` operation. The numbering in this paragraph is related to the numbering in the Figure. The `recv` call on a socket in the *pifus* backend may return immediately if there is already enough data in the buffer (1). Else, the `recv` operation is also put into queue (2) and popped later on when the lwIP callback has been invoked (3). If enough data has been received, the data is copied into the shared memory block given in the `recv` operation and the operation result is put into the completion queue (4, 5). The data is acked using `tcp_recved` for each operation that is successfully put into the completion queue and for data that has been buffered by the internal *pifus* `recv_buffer` (6).

The `accept` operation is also asynchronous, but does not use any queues to keep track of pending operations unlike the `write` or `recv` operations. When `accept` is invoked, lwIP's `tcp_accept` is called. When the corresponding callback is executed, a new entry to the completion queue is added. At the moment the client API dequeues this entry, it will automatically spawn a new socket region for this newly accepted socket. So unlike other operations, one `accept` operation in the submission queue can have multiple resulting entries in the completion queue.

### Driver

The *pifus* prototype is using the TAP interface that Linux offers. This is the reason why the stack thread uses `select` to get notified of new events on the interface.

In a real world scenario, *pifus* would use an interrupt based driver, for example one of the several existing ones for real hardware on lwIP. Generally, it is possible to use any other driver together with *pifus*. Not using TAP but an interrupt based driver would allow to remove the `eventfd` construct that is currently needed due to the `select` call when waiting for events.

### Quality of Service

*pifus* supports QoS mechanisms on a per-socket basis. As seen previously when the client API was presented, a socket can be assigned a priority. A priority in this case means that operations queued on a socket with a higher priority are always preferred to operations queued on a socket with a lower priority. This is in accordance with the requirements set by the analysis in Section 4.3.1.

The prototype of *pifus* supports three distinct and discrete priorities:

- `PRIORITY_HIGH`
- `PRIORITY_MEDIUM`
- `PRIORITY_LOW`

The relation between the priorities is therefore given by:

$$PRIORITY\_HIGH > PRIORIY\_MEDIUM > PRIORITY\_LOW$$

The implementation of the priority mechanism is as follows:

- Each *priority thread* is responsible for sockets of a certain priority. All operations queued on these sockets flow through this thread.
- All incoming operations of these sockets are put together in the *priority queue* as explained in the Subsection Design and Figure 5.8.
- The *stack thread* always tries to pop the *priority queue* with the highest priority first. Only after there are no elements in a higher priority queue, it tries to pop operations from a lower priority queue.

Using this algorithm, it is ensured that operations from a higher priority socket are always handled prior to operations from a lower priority socket.

Theoretically, *pifus* could be extended to an arbitrary amount of distinct priorities. But please note that for each priority a separate thread is launched, which may be unsuitable for a large amount of priorities depending on the system.

**Discovery**

The discovery implemented in *pifus* has the task of detecting new applications that are using the client API and also new sockets that are opened by these applications. It allows plug and play usage of *pifus* applications, as they are automatically connecting to the backend without the need of configuration by the user. This fulfills the goal set in the analysis in Section 4.1. The discovery process happens solely in the *discovery thread*.

As soon as a *pifus* client is started, it creates an application region as described in Section 5.1.4. The discovery thread regularly scans for new application regions by trying to open the next shared memory region following the `appX` name pattern, where `X` is the index of the next application. Assume that there are currently two open application regions:

1. `app0`
2. `app1`

Then the discovery would regularly (at the frequency of a user-defined constant) check if the shared memory region with the name `app2` exists, and if yes, would map it.

The socket discovery works instead on a futex basis. Each application region holds a futex which indicates the number of sockets that are opened. When a new socket is opened, this futex is increased and woken by the client. The socket discovery uses `futex_waitv` to wait on all these futexes across all application regions. Once one of the futexes is woken up, the socket discovery maps the new socket region(s) of the application.

The newly detected sockets are then passed into the `new_socket_queue` corresponding to the priority that the socket has.

Using `futex_waitv` for the detection of new sockets also implies that *pifus* supports at maximum 128 concurrent applications.

### TCP communication between pifus clients

When multiple *pifus* clients are connected to the same backend, they are also able to communicate with each other using TCP. An issue that can occur when two *pifus* clients are communicating with each other through the backend is a deadlock.

Assume one *pifus* client connects to another client and constantly writes some data using `write`, while the other client only `recv`s data from this connection. If the client writing the data issues many `write` operations so that the send buffer of lwIP runs full and there are still `write` operations on top of queue, while the `recv` operations of the other client are not yet dequeued because they are behind the `write` operations in the queue, a deadlock can occur. This is because no more data can be written due to the full send buffer, while the send buffer can not empty due to no `recv` operation being executed.

To mitigate this effect, `pifus` upper bounds the number of in-flight operations per socket. This does not fully mitigate the issue. Therefore, if *pifus* detects such a deadlock situation between two clients, it tries to scan the priority queues to find a `recv` operation that it can execute to solve the deadlock. From another perspective, bounding the number of in-flight operations also makes sure a malicious application may not overload the stack and therefore conforms to the goal set in Section 4.5.

### Configuration

*pifus* uses constants defined at compile time to e.g. set the size of queues. All relevant constants for tuning *pifus* are defined in `pifus_constants.h`. A non-exhaustive list of available constants is given by:

- sizes of the submission and completion queues
- sizes of the `write` and `recv` queues
- size of the `recv` buffer in bytes
- size of the priority queues
- amount of in-flight operations per socket
- patterns for the shared memory names

- frequency of the application discovery scanning

The client API and the backend must use the same configuration in order to be compatible.

## 5.2. Design space

Choosing the design presented in Section 5.1 was the result of several thoughts. These design choices concern further different aspects of *pifus*, such as the choice of the lwIP API to be used or how the priorities should be implemented.

### 5.2.1. Data structures

To minimize the end to end latency of operations, the data structures involved when operations flow through *pifus* (e.g. the ring buffer) have constant runtime in insertion and dequeing. The synchronization with futexes makes sure that the IPC latency is minimized, while other IPC mechanisms may introduce a larger latency.

Choosing the combination of constant data structures and fast synchronization between different processes and threads is a key point for reaching the goal of predictability.

### 5.2.2. lwIP API

There were several options on which lwIP API to use. lwIP offers three APIs, of which the raw API is the one on the lowest abstraction layer. [17] The netconn or the socket API offer a higher abstraction level, but also bound the user to specific usage patterns. For example, the netconn and socket API use extensive locking to ensure thread safety, while the raw API shifts the responsibility of avoiding race conditions to the user. This gives more freedom to the user to decide on which methods to use to achieve the goal of thread safety.

Furthermore, the socket and the netconn API are implemented using the raw API. Therefore, they do not offer more functionality than the raw API does, but possibly only a subset of functionality.

Simply put, the raw API offers as much freedom as possible to use the lwIP stack, while consuming as less resources as possible by e.g. not offering built-in thread safety. Therefore, it was decided that *pifus* should use the raw API of lwIP.

Designs using the netconn or socket API in contrast would have been bound to lwIPs threading model. In non-blocking mode of the netconn sockets for example, this would mean that the amount of threads linearly scales with the amount of sockets.

### 5.2.3. lwIP mode

Another possibility that lwIP offers is to decide between the OS mode and the mainloop mode [34]. In contrast to the mainloop mode, in the OS mode lwIP automatically starts its `tcpip-thread` over which the user does not have any control. The `tcpip-thread` automatically handles TCP timers and driver polling, while in the mainloop mode the user has to invoke the corresponding methods.

We chose to use the mainloop mode, so that there would not be another thread besides the *stack thread* and therefore to keep the number of threads and communication between them minimal.

### 5.2.4. Priority threads

For *pifus*, it was decided to create a distinct thread for each priority. While this imposes a larger thread count (especially when more priority levels are added), it mitigates the issue that the `futex_waitv` system call only supports to wait for up to 128 futexes. [30]

Furthermore, it is assured that the waking of futexes for sockets that are of lower priority does not impact higher priority sockets.

An alternative design would be to use one thread for all priorities. While this design imposes a smaller thread count, it also limits the amount of sockets to 127 and maybe leads to issues respecting the priorities of sockets.

Another alternative would be to have a pool of threads, where each thread in the pool is responsible for multiple priority levels. For a large amount of distinct priority levels, this would result in less threads than in the original design. This would probably be the best approach if there are many priority levels. An open question would be on how the priority levels are assigned to the particular priority threads to achieve a good split of load between the threads.

### 5.2.5. Application multiplexing

Application multiplexing, as required by Section 4.3.5, can be solved in multiple ways. *pifus* tackles application multiplexing by introducing priority threads, of which each is responsible for a set of sockets of equal priority. This way, *pifus* mitigates the issue of doing multiplexing on three levels, namely the applications, the sockets, and the priorities. *pifus* instead simplifies the multiplexing to just one level by keeping a list of sockets with equal priority, irrespective of their application affiliation.

Alternatives to this approach, for example doing the multiplexing on the application layer instead of the socket layer, may lead to a design that is more complex due to having to multiplex on several layers. Additionally, such a design may disrespect the predictability requirements set in Section 4.3.1 because the predictability requirements are violated if the first layer of multiplexing does not respect the priorities of the sockets, but other properties of the socket, such as the application affiliation of the socket.

### 5.2.6. Memory management

Regarding dynamic memory allocation, there are alternatives to the use of shared memory with a memory management algorithm as it is implemented in *pifus*.

For example, one alternative could be *anonymous files* [36]. *Anonymous files* can be mapped into memory by different processes and then be used to share data between these processes. For each allocation in *pifus*, a different *anonymous file* could be used. Then the `alloc` and `free` memory management functions would not be necessary, but a system call would be introduced for every memory allocation. Furthermore, *anonymous files* are Linux-specific and therefore not as easily portable as POSIX shared memory.

Another option would be the usage of `process_vm_readv` and `process_vm_writev` [37] to transfer data between processes. These are also Linux-specific system calls and would require a system call per data (de)allocation. A similar functionality is also available on other operating systems, such as in macOS with the Mach microkernel API using `mach_vm_read` and `mach_vm_write` [38, Sections 8.6.7, 8.6.8].

A rather different approach would be to use ptrace [39] to read and write memory to a different process memory space.

In the end, the decision to use shared memory combined with own `alloc` and `free` methods was taken because the alternatives are bound to Linux and require more system calls. Furthermore, as shared memory is already used for the socket region, it is also consistent to use it for the allocation of dynamic memory that is shared between the processes.

## 5.3. Portability

The current *pifus* prototype was designed to be run on Linux.

Theoretically, *pifus* could be ported to any operating systems that support POSIX [27], mainly due to the usage of POSIX shared memory [28] and `pthreads`. This is in accordance with the analysis requirement set in Section 4.6.

Another point that *pifus* is inherently dependent upon is the usage of futexes. The target system should therefore also support some kind of synchronization mechanisms similar to futexes, especially with the possibility to wait on multiple futexes. For Linux this implies that at least kernel version `5.16` is required.

The lwIP stack itself is designed to be able to run basically on any system and therefore does not impose any restrictions regarding portability.

# 6. Evaluation

The following subsections evaluate *pifus* regarding the requirements set in Chapter 4. In Section 6.1, *pifus* is evaluated experimentally using benchmarks, while Section 6.3 takes a theoretical approach. Section 6.2 analyzes the implementation of *pifus* using tracing tools and tries to find bottlenecks.

## 6.1. Experimental evaluation

The experimental evaluation has been conducted using benchmarks. *pifus* was built with compiler optimizations (`-O2`) and run on a Linux system with kernel version `5.16.0-051600-generic`. The kernel was not patched, so e.g. no real-time patch was applied to it.

For some measurements, CPU affinities where set using `taskset`. If not explicitly mentioned, no affinity was set for the particular benchmark.

The hardware of the machine where the benchmarks were run was:

- Intel i7-2600 @ 4x3.40GHz
- 16 GB RAM

All the benchmarks are focused on latency as metric, as predictability is the main concern of *pifus*. Performance on the other hand is not the focus of *pifus*.

### 6.1.1. Communication latency

The communication latency of *pifus* is the time between the enqueuing of an operation and the dequeuing of the finished operation using the client API. To effectively measure this latency without the influence of lwIP and the network, a `NOP` operation has been introduced. When dequeued by the backend, this `NOP` operation is instantly inserted into the completion queue. This way, only the overhead of the IPC between the client and the stack is measured.

The latency is then calculated using the following formula:

$$latency = t_{completed} - t_{submitted}$$

Where $t_{completed}$ denotes the timestamp when the operation is dequeued from the completion queue and $t_{submitted}$ denotes the timestamp when the operation was enqueued into the submission queue.

For this benchmark, the poll mode of the *pifus* API is used. A client is started and submitting a `NOP` operation. After it received the result of a `NOP` operation, the next one is enqueued. This continues until 30 seconds have passed and the benchmark is completed.

A scatterplot depicting the communication latency measurements is given by Figure 6.1, while descriptive statistics of the measurements are presented in Table 6.1.

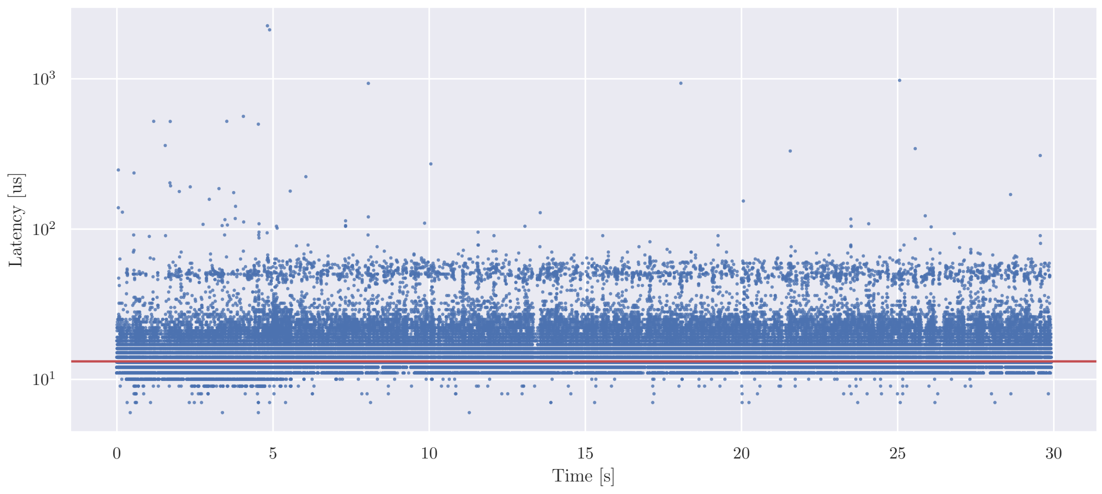

Figure 6.1.: Scatterplot of communication latency. The red line indicates the average.

| Mean | Stdev | Min | 25% | 50% | 75% | Max |
|---|---|---|---|---|---|---|
| 13.2 | 3.3 | 6.0 | 12.0 | 13.0 | 14.0 | 2236.0 |

Table 6.1.: Descriptive statistics of the communication latency measurement data in µs.

The communication latency of *pifus* is centered around 13 µs with only a small standard deviation. A small standard deviation indicates that the latency jitter is low, which is a desirable for meeting predictability goals as defined in Section 4.3. The large maximum observation in the measurements can be explained with the operating system scheduler, which may choose to schedule a different process as we use no real-time patch.

### 6.1.2. Priority comparison

To evaluate the *pifus* QoS mechanisms, a benchmark with three competing sockets of different priorities was taken. For this, three applications with a socket each of different priority using the callback mode of the *pifus* client API were started. Every application executes `NOP` operations and keeps a certain amount of these operations in flight. The latencies are calculated as it was done for the benchmark in Section 6.1.1.

The applications are set to run on a different core each by using `taskset`. The *pifus* backend is pinned to a separate core.

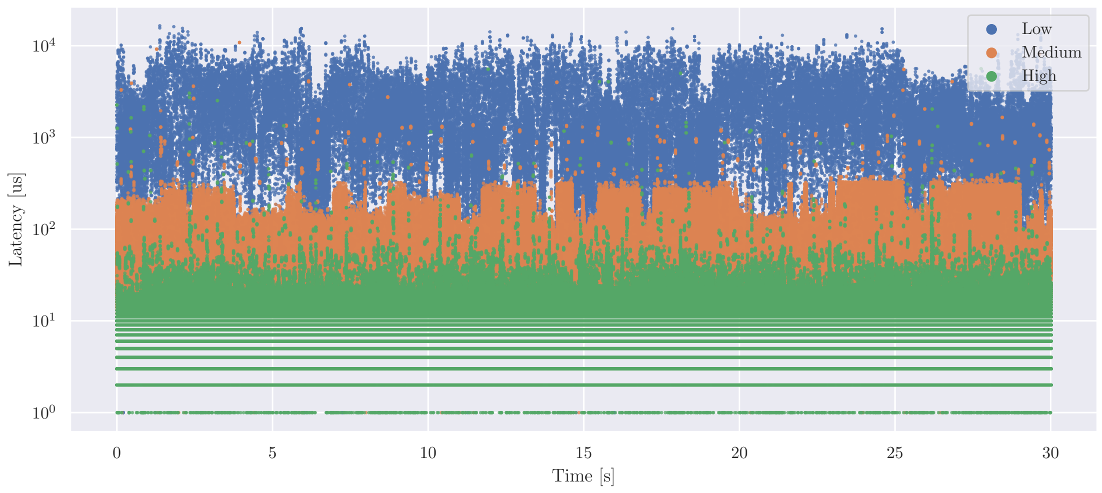

Figure 6.2.: Scatterplot of latencies using sockets with different priorities.

Figure 6.2 shows that the operations executed on the higher priority sockets have smaller latencies than the ones executed on the lower priority sockets. This shows that the mechanism meets the expectations defined in Section 4.3, namely that the prioritization of sockets is possible in a predictable way.

This can also be seen when looking at Table 6.2. The mean latency values are roughly a magnitude higher for lower priority sockets when comparing them to a higher priority socket. The same holds true for the standard deviation.

| Priority | Mean | Stdev | Min | 25% | 50% | 75% | Max |
|---|---|---|---|---|---|---|---|
| High | 6.4 | 7.7 | 0.0 | 5.0 | 6.0 | 8.0 | 5578.0 |
| Medium | 58.2 | 50.3 | 1.0 | 30.0 | 55.0 | 75.0 | 10765.0 |
| Low | 927.7 | 1431.1 | 1.0 | 17.0 | 382.0 | 1199.0 | 16347.0 |

Table 6.2.: Descriptive statistics of the priority comparison measurement data in µs.

One noteworthy observation in the Table may be the low minimum values. These can occur when the callback thread of the client API is woken up because a new operation has been inserted into the completion queue. In the meantime, other operations may have also finished and are dequeued. Therefore, some operations in the batch bypass the most costly process of futex operations (`wake` and `wait`) and consequently have a small latency.

A similar scenario was also benchmarked, but this time with data on the wire. Instead of a `NOP` operation that is executed, three applications with each a socket `connect` and `write` to another non-*pifus* application which just calls `recv` and acknowledges the data sent.

The TAP interface of Linux was used for the data transmission between the applications. The results of this benchmark can be seen in Table 6.3.

| Priority | Mean | Stdev | Min | 25% | 50% | 75% | Max |
|---|---|---|---|---|---|---|---|
| High | 144.9 | 176.1 | 34.0 | 118.0 | 135.0 | 152.0 | 12890.0 |
| Medium | 160.9 | 172.8 | 38.0 | 128.0 | 150.0 | 177.0 | 11955.0 |
| Low | 163.6 | 187.1 | 30.0 | 131.0 | 151.0 | 176.0 | 11955.0 |

Table 6.3.: Descriptive statistics of the priority comparison on the wire measurement data in µs.

Similar to the previous benchmark, the sockets with a higher priority mostly have smaller latencies in the categories depicted in Table 6.3. The difference is not as clear as it is with the previous benchmark, but this may be due to the data actually being sent on the wire and the applications only keeping a certain amount of operations in flight. Therefore, when all higher priority operations have already been executed and are waiting for the remote side to be acknowledged, the lower priority operations can be executed. For the `NOP` operations this is not the case to this extent, as the execution time of the `NOP` operations is very small by design to only measure the communication overhead of the *pifus* interface.

Taking the benchmarks into consideration, it can be said that the prioritization in *pifus* performs as required in Chapter 4. Sockets with a higher priority effectively had lower latencies compared to lower priority sockets.

### 6.1.3. Comparison with lwIP netconn API

Similar to *pifus*, the lwIP netconn API offers the possibility to use lwIP from multiple threads. This is realized with locking for netconn. This benchmark compares the latencies of *pifus* and netconn when concurrently using multiple sockets on different threads. The benchmark is using network operations on the wire. For each *pifus* and netconn run, a different amount of threads, of which each is opening a socket, is spawned. The resulting sockets `connect` to another application and call `write`, similar to the wire benchmark in Section 6.1.2. Each run takes 30 seconds. The latency is then obtained the same way as in Section 6.1.1.

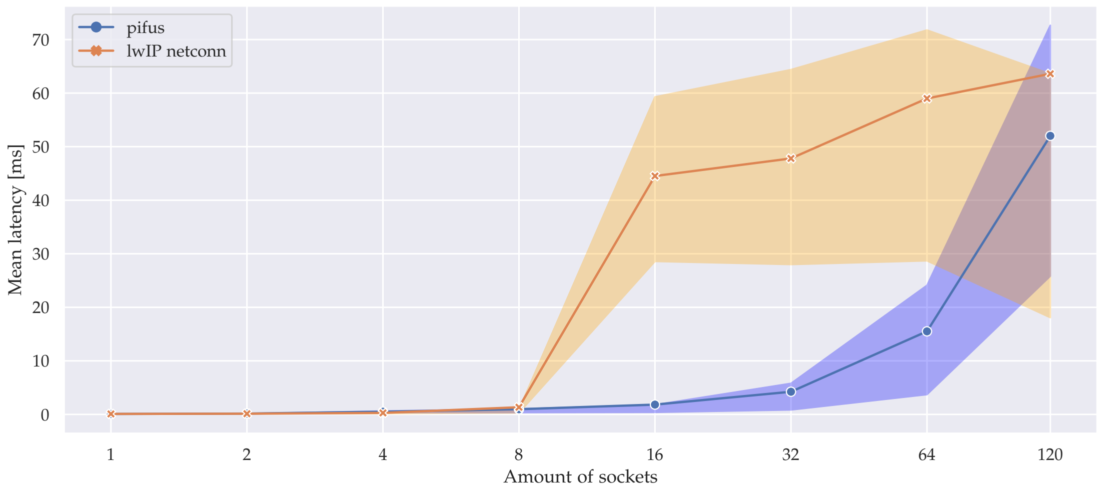

Figure 6.3.: Line plot of the mean latencies with 25% and 75% quantiles.

Figure 6.3 shows the averaged latencies across all sockets of the particular run. For a small number of sockets (one to eight), *pifus* and netconn have similar mean latencies. For a larger number of sockets than eight, the mean latencies and interquartile ranges for *pifus* are significantly lower than they are for netconn. This may be because above this thread count, the locking mechanisms of netconn introduce delays due to too many threads competing for the lwIP stack.

Table 6.4 contains more detailed statistics about the latencies collected in the benchmark. Despite what could be seen in the Figure already, the Table shows that for a small number of sockets (< 8), *pifus* performs a bit worse than netconn. This may be because *pifus* needs IPC, while for netconn the network stack resides in the same process and no IPC is needed, just the synchronization for the different threads. For a larger number of sockets ($\geq 8$), this effect is heavily outweighed by the overhead caused from locking in netconn and this makes *pifus* performs better than netconn.

| # Sockets | API | Mean | Median | Min | Max | 25% | 75% |
|---|---|---|---|---|---|---|---|
| 1 | netconn | 81.29 | 81.0 | 35.0 | 21905.0 | 68.0 | 88.0 |
| | *pifus* | 85.6 | 80.0 | 31.0 | 15944.0 | 70.0 | 88.0 |
| 2 | netconn | 129.75 | 117.0 | 39.0 | 15940.0 | 96.0 | 148.0 |
| | *pifus* | 150.01 | 144.0 | 43.0 | 17585.0 | 131.0 | 156.0 |
| 4 | netconn | 276.35 | 226.0 | 40.0 | 12506.0 | 159.0 | 319.0 |
| | *pifus* | 504.40 | 462.0 | 60.0 | 20932.0 | 343.0 | 629.0 |
| 8 | netconn | 1338.55 | 482.0 | 40.0 | 53853.0 | 280.0 | 937.0 |
| | *pifus* | 974.42 | 640.0 | 60.0 | 36415.0 | 337.0 | 1045.0 |
| 16 | netconn | 44517.31 | 43844.0 | 52.0 | 226395.0 | 28569.5 | 59298.5 |
| | *pifus* | 1831.83 | 809.0 | 51.0 | 62228.0 | 391.0 | 1795.0 |
| 32 | netconn | 47801.60 | 46660.5 | 49.0 | 447535.0 | 27993.25 | 64389.5 |
| | *pifus* | 4227.19 | 2861.0 | 50.0 | 90536.0 | 837.0 | 5797.0 |
| 64 | netconn | 59001.74 | 48185.0 | 45.0 | 2272040.0 | 28684.5 | 71775.5 |
| | *pifus* | 15513.16 | 11707.0 | 70.0 | 103108.0 | 3704.0 | 24110.0 |
| 120 | netconn | 63605.46 | 39713.0 | 53.0 | 2760155.0 | 18065.0 | 63611.5 |
| | *pifus* | 52021.89 | 48743.0 | 70.0 | 263665.0 | 25823.0 | 72665.75 |

Table 6.4.: Descriptive statistics of the netconn baseline measurement data in µs.

Generally looking at the benchmark results, *pifus* scales well with an increasing number of sockets, while netconn saturates on a high level. *pifus* therefore reached the goal of combining scalability and predictability defined in Chapter 4 in comparison to already existing lwIP APIs.

## 6.2. Tracing

`perf` [40] was used as a tracing tool to find hotspots in the code which may take much CPU time.

Figure 6.4 shows the flame graph of the *pifus* backend with a client connected that executes `NOP` operations. By using only `NOP` operations, CPU time taken by TAP operations or the network in general are not included in the flame graph and the *pifus* code can be analyzed without this interference. In the Figure, it can be seen that most time is spent in system calls such as `select`, `open`, `read` and `write`. These system calls are mainly used for driver interaction as explained in Section 5.1.5. With real hardware and an interrupt based driver, these system calls would not be necessary anymore. Other system calls taking CPU time seen in the flame graph are the `futex_waitv` and `futex_wake` system calls.

*pifus* internal functions only account for a small part of the Figure, indicating that they do not consume as much CPU time as the system calls. This means there is no bottleneck observable in the backend which could be easily optimized.

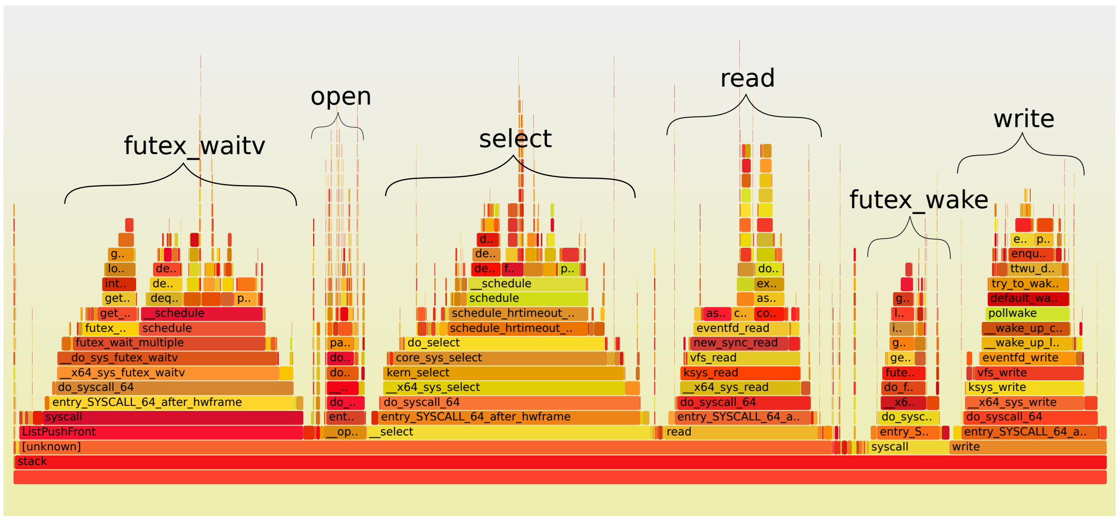

Figure 6.4.: Perf flame graph of the *pifus* backend when a client is constantly queuing `NOP` operations.

Similarities can be seen when looking at Figure 6.5. Much time is spent for the system calls `futex_waitv` and `futex_wake`. Furthermore, the `main` method is taking much time because it is trying to insert `NOP` operations into the submission queue in a `while` loop to keep a certain amount of operations in flight. The CPU time spent in the `pifus` API methods is very low again and barely observable in the Figure, e.g. the method `pifus_socket_nop`, which inserts a `NOP` operation into the submission queue, only accounts for 0.3% of CPU time.

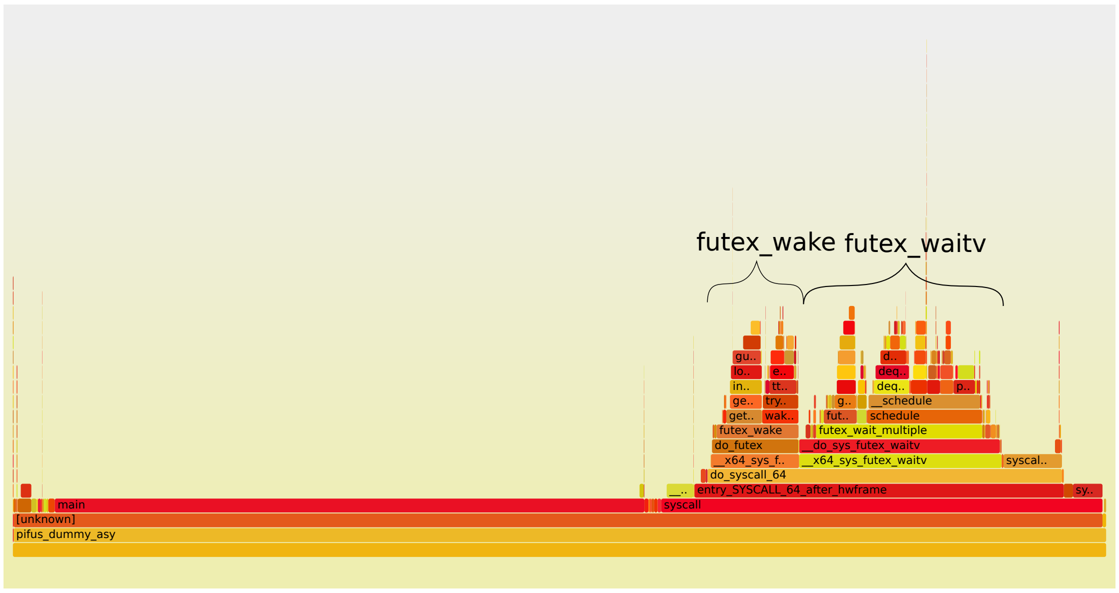

Figure 6.5.: Perf flame graph of the *pifus* client using the callback mode and queuing `NOP` operations.

To summarize, there were no apparent bottlenecks when speaking about CPU time in the client API as well as in the backend.

## 6.3. Theoretical evaluation

Subsection 6.3.1 evaluates on a theoretical basis how the latencies of *pifus* behave when enqueuing and dequeuing operations. Subsection 6.3.2 gives a perspective of how *pifus* applications interfere with each other.

### 6.3.1. Latency

#### Enqueuing operations

When submitting an operation via the *pifus* client API, the general flow of the operation is:

$$\mathit{Client\ API} \xrightarrow{\texttt{put}\ (1)} \mathit{submission\ queue} \xrightarrow{\texttt{get}\ (2),\ \texttt{put}\ (3)} \mathit{priority\ queue} \xrightarrow{\texttt{get}\ (4)} \mathit{lwIP\ API}$$

Where the submission queue and the priority queue are both ring buffers as presented in Section 5.1.3. Additionally, as these buffers are distributed across process boundaries, a futex is used for synchronization between step (1) and (2).

When using the futex, a `futex_wake` and a `futex_waitv` system call is used. As a detailed analysis of these system calls is out of scope for this thesis, we look at the system calls as if they were a black box. For `futex_wake` we approximate a time complexity of $O(1)$, as looking up a futex and waking it should be possible in constant time.

In contrast, we approximate `futex_waitv` to be of $O(n)$ time complexity, where $n$ is the number of submission queues, which equals the number of sockets, that the backend waits for. The approximation of a linear time complexity for `futex_waitv` is a result of the assumption that internally, a list of futexes to wait on has to be hold, and when a single futex is woken, one has to iterate over the list to notice which futex was woken, thus resulting in $O(n)$ time complexity depending on the number of futexes to wait on.

The ring buffer operations used (`put` and `get`) are both of $O(1)$ time complexity. See Algorithm 1 and Algorithm 2 for reference on why this is the case.

Let $T_{operation}$ be the time complexity of an operation. Then, when looking at the total time complexity of enqueuing an operation, the following holds true:

$$\begin{aligned} T_{put(1)} + T_{futex\_wake} + T_{futex\_waitv} + T_{get(2)} + T_{put(3)} + T_{get(4)} \\ = O(1) + O(1) + O(n) + O(1) + O(1) + O(1) \\ = O(n) \end{aligned}$$

where $n$ is the number of sockets of a certain priority that are connected to the backend.

It is observable that the bottleneck in terms of time complexity is the `futex_waitv` system call, as it linearly scales with the amount of connected sockets. The *pifus* data structures on the other hand have constant time complexities.

### Dequeuing operations

The flow for dequeing finished operations from the completion queue looks like that:

$$\textit{Client API} \xleftarrow{\texttt{get (2)}} \textit{completion queue} \xleftarrow{\texttt{put (1)}} \textit{lwIP API}$$

Similar as previously, we need a futex for synchronization as the ring buffer is used across process bounds. In this case, the futex is woken in step (1). As seen in Section 5.1.4, in the callback mode the client API uses `futex_waitv` to wait for incoming operations on the completion queues of the sockets. In the poll mode, the user may use the `futex_wait` system call. For the calculation of the time complexities, we have to distinguish these two cases, as different to the `futex_waitv` call we can approximate the time complexity of the `futex_wait` call with $O(1)$.

Therefore, when using the callback mode, the time complexity is given by:

$$\begin{aligned}
&T_{put(1)} + T_{futex\_wake} + T_{futex\_waitv} + T_{get(2)} \\
&= O(1) + O(1) + O(n) + O(1) \\
&= O(n)
\end{aligned}$$

In contrast, when waiting for results on a single socket by using the poll mode in the client API, the time complexity is given by:

$$\begin{aligned}
&T_{put(1)} + T_{futex\_wake} + T_{futex\_wait} + T_{get(2)} \\
&= O(1) + O(1) + O(1) + O(1) \\
&= O(1)
\end{aligned}$$

By using the poll mode, one can mitigate the time complexity that the `futex_waitv` system call imposes. Of course, this brings the disadvantage of being only woken when an operation finished on this specific socket.

**Summary**

To summarize, the latency when enqueuing and receiving operations heavily depends on the `futex_waitv` system call. This system call multiplexes multiple I/O paths, i.e. is the key point in combining paths from multiple sockets to one processing pipeline in the stack. As this implies listening for notifications on at least $n$ paths, it is clear that this method generally needs to have at least a linear runtime. Therefore, this shortcoming is not bound to the explicit usage of `futex_waitv`, but a general result of merging several communication paths into one path.

### 6.3.2. Interference

Another theoretical perspective can be made on how applications using *pifus* interfere with each other. As seen in Section 5.1.1, *pifus* creates a separate shared memory region for every application. This means that the dynamically allocated data stored in this region is also only accessible for this certain application. This is important to fulfill the goal of data confidentiality set in Section 4.5.

Furthermore, this separation also ensures that there is no predictability degradation. As mentioned in Section 5.1.3, the memory allocation algorithm used for managing this region is not lock-free and requires coordination. Without the separation, predictability degradation could happen if multiple applications want to dynamically allocate memory inside the region. Then, a global lock would be needed, which may increase the latencies.

This separation can also be seen when looking at the implementation of the sockets. As seen in Section 5.1.1, for each socket a region holding data structures is allocated. Each socket has its separate data structures, e.g. a separate submission and completion queue. This implies that the sockets are independent of each other, meaning that queuing an operation in one socket does not influence the data structures of another socket, achieving the goals set in Section 4.3.4. Of course, the submission queues are merged in the backend based on their priority later, but sockets do not interfere when looking at the communication between the client API and the backend.

Generally put, *pifus* does not use any global locks and does not require any communication or coordination between different applications using the *pifus* client API.

To summarize, *pifus* tries to separate I/O paths by using separate data structures. This helps to reach the goal of predictability, as less interference leads to higher predictability as defined in Section 4.3.

# 7. Conclusion

In this thesis, a predictable interface for a user space IP stack has been proposed. Specifically, lwIP is the identified user space network stack that has been extended with the proposed interface. The selection of the IP stack was elaborated, design decisions made have been explained, and the interface was evaluated in theory as well as in practice.

The main objective when developing the interface was predictability in a sense that real-time applications are preferred over non real-time applications and furthermore that the interface induces a low communication latency.

The developed interface (*pifus*) has been shown to meet the expectations by solving the problem of connecting multiple applications to a user space IP stack in a predictable way by offering QoS mechanisms that allow prioritization of real-time applications using the network stack. Furthermore, *pifus* achieves predictability by offering a socket API with low communication latency overhead to the network stack and mitigates issues that can occur with synchronous I/O multiplexing by design due to the usage of completion ports.

The evaluation part of the thesis further showed that *pifus* outperformed the lwIP netconn API significantly when more than eight sockets are active in parallel. The evaluation part also confirmed that the QoS mechanisms in *pifus* lead to real-time applications having lower latencies than non real-time applications.

Of course, this thesis also has certain limitations. These are outlined in Section 7.1. Moreover, suggestions for possible future work that could be done regarding the topic are made in Section 7.2.

## 7.1. Limitations

The limitations of this thesis can be divided into two areas, namely the limitations that are present in the *pifus* implementation and limitations regarding the evaluation part of the thesis.

One limitation of the *pifus* implementation is that the discovery described in Section 5.1.5 is prone to race conditions when multiple applications connect to the *pifus* stack simultaneously. Other competing actions, such as for example concurrent socket creation by multiple applications, are not effected by this issue due to the design of *pifus*.

The QoS implementation of *pifus* has several limitations:

- The mechanism is solely based on priorities of sockets.
- While preallocation of data for periodic real-time flows was not yet implemented, introducing it could further improve predictability by reducing latencies induced by the allocation of memory blocks inside shared memory.
- The queue sizes are defined on compile time and therefore fixed, instead of dynamic queue sizes that depend on socket-specific QoS settings.

Furthermore, the memory allocation algorithm described in Section 5.1.3 is not optimal in terms of its time complexity and memory fragmentation, due to using a first fit approach.

Another limitation is that pipelining of operations is not supported. For example, when enqueuing an operation, pipelining allows specifying which other operation has to finish before the enqueued operation. To summarize, pipelining offers a way to influence the order of operations. In the current implementation, *pifus* handles operations in FIFO fashion, while, for example, `io_uring` supports pipelining [41].

Regarding the evaluation limitations, it can be said that a benchmark comparing synchronous I/O multiplexing with *pifus* completion ports would be interesting. This thesis did not provide such a benchmark because setting up such a benchmark is not trivial and was therefore out of scope and not strictly required for the evaluation of the main predictability properties of the interface.

## 7.2. Future Work

Future work on the topic could try to lift some limitations mentioned in Section 7.1.

For example, the discovery could be made thread-safe using another communication channel prior to the creation of shared memory regions. This could be realized with POSIX message queues or Unix sockets to coordinate the creation of shared memory regions. This communication channel could also be used for implementing further QoS aspects, such as queue sizes that are configurable on a socket level. In this case, the communication channel would be needed because the size of the shared memory region is adjusted based on the queue sizes and needs to be communicated to the *pifus* backend.

In addition, the preallocation or reuse of memory blocks for periodic real-time data could completely be implemented on the client API side by not freeing memory blocks when the corresponding operation has finished, but keeping them allocated and usable for periodic operations such as for example a regular ping that embedded devices usually send.

Furthermore, pipelining could be implemented by assigning each operation a unique identifier. Then, a check in the backend would be necessary to ensure that the operation is executable without violating the pipelining constraints.

The memory allocation algorithm of *pifus* could be improved by using a best fit or next fit approach instead of the first fit approach. Especially the next fit approach would reduce latencies when repeatedly searching for a suitable memory block to use. Other algorithms or preallocation of memory could also be considered.

Regarding the evaluation part of this thesis, further benchmarks could be run. Especially benchmarks that compare synchronous I/O multiplexing with completion ports would be a good addition to assess the trade-offs between the two approaches.

Moreover, benchmarks could also be run on another environment, e.g. on an embedded board with ARM architecture to verify *pifus* properties and performance on embedded hardware.

Another perspective on the benchmarks could also be given by using *pifus* on real networks instead of just on virtual interfaces like TAP. To do this, drivers for the NIC have first to be integrated into *pifus*. Then, the benchmarks could be run on embedded hardware.

# A. Code Repository

The code of *pifus* is available at the following Git repository:
`https://gitlab.lrz.de/chair-of-cyber-physical-systems-in-production-engineering/thesis/oliver_layer_pifus`

The repository contains lwIP and all components of *pifus*, such as the client API and the backend. Furthermore, examples of applications using *pifus* can be found in the repository. For further information, please refer to the `README.md` file in the repository root.

# Bibliography

[1] *select(2) — Linux manual page*, `https://man7.org/linux/man-pages/man2/select.2.html`, last accessed on Apr. 2022.

[2] *poll(2) — Linux manual page*, `https://man7.org/linux/man-pages/man2/poll.2.html`, last accessed on Apr. 2022.

[3] *epoll(7) — Linux manual page*, `https://man7.org/linux/man-pages/man7/epoll.7.html`, last accessed on Apr. 2022.

[4] *Select is fundamentally broken*, `https://idea.popcount.org/2017-01-06-select-is-fundamentally-broken/`, last accessed on Apr. 2022, 2017.

[5] *Efficient IO with io_uring*, `https://kernel.dk/io_uring.pdf`, last accessed on Apr. 2022.

[6] *aio(7) — Linux manual page*, `https://man7.org/linux/man-pages/man7/aio.7.html`, last accessed on Apr. 2022.

[7] D. McCall, *Asynchronous I/O on linux*, `http://davmac.org/davpage/linux/async-io.html`, last accessed on Apr. 2022.

[8] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum, "Keep net working - on a dependable and fast networking stack," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12. DOI: `10.1109/DSN.2012.6263933`.

[9] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing network protocols at user level," *SIGCOMM Comput. Commun. Rev.*, vol. 23, no. 4, pp. 64–73, Oct. 1993, ISSN: 0146-4833. DOI: `10.1145/167954.166244`.

[10] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda, "IsoStack—Highly efficient network processing on dedicated cores," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, USENIX Association, Jun. 2010.

[11] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: A microkernel approach to host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 399–413, ISBN: 9781450368735. DOI: `10.1145/3341301.3359657`.

[12] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack," *Swedish Institute of Computer Science*, vol. 2, Mar. 2001.

[13] A. Dunkels and L. Woestenberg, *lwIP Lightweight IP stack*, `http://www.nongnu.org/lwip/2_1_x/index.html`, last accessed on Apr. 2022.

[14] lwIP Wiki, *Projects that use lwIP*, `https://lwip.fandom.com/wiki/Projects_that_use_lwIP`, last accessed on Apr. 2022.

[15] ——, *Available device drivers*, `https://lwip.fandom.com/wiki/Available_device_drivers`, last accessed on Apr. 2022.

[16] ——, *lwIP Application Developers Manual, Application API layers*, `https://lwip.fandom.com/wiki/Application_API_layers`, last accessed on Apr. 2022.

[17] non-gnu lwIP Wiki, *lwIP APIs*, `http://www.nongnu.org/lwip/2_1_x/group__api.html`, last accessed on Apr. 2022.

[18] ——, *lwIP "raw"APIs*, `http://www.nongnu.org/lwip/2_1_x/group__callbackstyle__api.html`, last accessed on Apr. 2022.

[19] lwIP Wiki, *lwIP Application Developers Manual, Netconn*, `https://lwip.fandom.com/wiki/Netconn_API`, last accessed on Apr. 2022.

[20] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "Mtcp: A highly scalable user-level TCP stack for multicore systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502, ISBN: 978-1-931971-09-6.

[21] mTCP Team, *mTCP GitHub Repository*, `https://github.com/mtcp-stack/mtcp`, last accessed on Apr. 2022.

[22] A. E. Belgium, *picoTCP GitHub Repository*, `https://github.com/tass-belgium/picotcp`, last accessed on Apr. 2022.

[23] ——, *picoTCP GitHub Release Page*, `https://github.com/tass-belgium/picotcp/releases/`, last accessed on Apr. 2022.

[24] Tencent Cloud, *F-Stack | High Performance NEtwork Framework Based on DPDK*, `http://www.f-stack.org/`, last accessed on Apr. 2022.

[25] smoltcp Team, *smoltcp GitHub Repository*, `https://github.com/smoltcp-rs/smoltcp`, last accessed on Apr. 2022.

[26] S. Lankes, J. Klimt, J. Breitbart, and S. Pickartz, "Rustyhermit: A scalable, rust-based virtual execution environment," in *High Performance Computing*, H. Jagode, H. Anzt, G. Juckeland, and H. Ltaief, Eds., Cham: Springer International Publishing, 2020, pp. 331–342, ISBN: 978-3-030-59851-8.

[27] "IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))," *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pp. 1–3874, 2008. DOI: `10.1109/IEEESTD.2008.4694976`.

[28] *shm_overview(7) — Linux manual page*, `https://man7.org/linux/man-pages/man7/shm_overview.7.html`, last accessed on Apr. 2022.

[29] *futex(2) — Linux manual page*, `https://man7.org/linux/man-pages/man2/futex.2.html`, last accessed on Apr. 2022.

[30] A. Almeida, *futex2 - The Linux Kernel documentation*, `https://www.kernel.org/doc/html/latest/userspace-api/futex2.html`, last accessed on Apr. 2022.

[31] QuantumLeaps, *Lock-free ring buffer GitHub repository*, `https://github.com/QuantumLeaps/lock-free-ring-buffer`, last accessed on Apr. 2022.

[32] Z. Shen, *C common data structures GitHub repository*, `https://github.com/ZSShen/C-Common-Data-Structures`, last accessed on Apr. 2022.

[33] *eventfd(2) — Linux manual page*, `https://man7.org/linux/man-pages/man2/eventfd.2.html`, last accessed on Apr. 2022.

[34] non-gnu lwIP Wiki, *lwIP: Mainloop mode*, `https://www.nongnu.org/lwip/2_1_x/group__lwip__nosys.html`, last accessed on Apr. 2022.

[35] ——, *lwIP "raw"API TCP documentation*, `https://www.nongnu.org/lwip/2_1_x/group__tcp__raw.html`, last accessed on Apr. 2022.

[36] *memfd_create(2) — Linux manual page*, `https://man7.org/linux/man-pages/man2/memfd_create.2.html`, last accessed on Apr. 2022.

[37] *process_vm_readv(2) — Linux manual page*, `https://man7.org/linux/man-pages/man2/process_vm_readv.2.html`, last accessed on Apr. 2022.

[38] A. Singh, *Mac OS X Internals*. Boston, MA: Addison-Wesley Educational, Jan. 2016.

[39] *ptrace(2) — Linux manual page*, `https://man7.org/linux/man-pages/man2/ptrace.2.html`, last accessed on Apr. 2022.

[40] *Perf wiki*, `https://perf.wiki.kernel.org/index.php/Main_Page`, last accessed on Apr. 2022.

[41] J. Axboe, *io_uring: support for linked SQEs*, `https://lwn.net/Articles/788929/`, last accessed on Apr. 2022.