Rust and Inter-Process Communication (IPC) on L4Re

Implementing a Safe and Efficient IPC Abstraction

Name	Sebastian Humenda
Supervisor	Dr. Carsten Weinhold
Supervising Professor	Prof. Dr. rer. nat. Hermann Härtig
Published at	Technische Universität Dresden, Germany
Date	11th of May 2019

Contents

1 Introduction **5**

2 Fundamentals **8**
2.1 L4Re . . . 8
2.2 Rust . . . 11
2.2.1 Language Basics . . . 11
2.2.2 Ecosystem and Build Tools . . . 14
2.2.3 Macros . . . 15

3 Related Work **17**
3.1 Rust RPC Libraries . . . 17
3.2 Rust on Other Microkernels . . . 18
3.3 L4Re IPC in Other Languages . . . 20
3.4 Discussion . . . 21
3.4.1 Remote Procedure Call Libraries . . . 21
3.4.2 IDL-based interface definition . . . 22
3.4.3 L4Re IPC Interfaces In Other Programming Languages . . . 23

4 L4Re and Rust Infrastructure Adaptation **24**
4.1 Build System Adaptation . . . 25
4.1.1 Libraries . . . 26
4.1.2 Applications . . . 27
4.2 L4rust Libraries . . . 29
4.2.1 L4 Library Split . . . 29
4.2.2 Inlining vs. Reimplementing . . . 30
4.3 Rust Abstractions . . . 31
4.3.1 Error Handling . . . 31
4.3.2 Capabilities . . . 32

5 IPC Framework Implementation **35**
5.1 A Brief Overview of the C++ Framework . . . 36
5.2 Rust Interface Definition . . . 37
5.2.1 Channel-based Communication . . . 37
5.2.2 Macro-based Interface Definition . . . 39
5.3 Data Serialisation . . . 43

5.4 Server Loop . . . 46
5.4.1 Vector-based Service Registration . . . 46
5.4.2 Pointer-based Service Registration . . . 48
5.5 Interface Export . . . 52
5.6 Implementation Tests . . . 54

6 Evaluation **57**
6.1 Execution Performance . . . 57
6.1.1 Primitive Types . . . 59
6.1.2 Strings . . . 60
6.1.3 Round Trip Measurements . . . 62
6.2 Usability . . . 63
6.2.1 Safety . . . 64
6.2.2 Interface Usage . . . 64
6.2.3 Stable Rust . . . 66
6.2.4 L4Re Integration . . . 66

7 Conclusion **67**

A References **69**

B Glossary **72**

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Als Quelle dienten ausschließlich die im Literaturverzeichnis aufgeführten Quellen, die im Text durch Zitate kenntlich gemacht wurden.

Dresden, 11.05.2019

1 Introduction

The ever increasing complexity of operating systems led to large code bases and complex interactions between their components. Traditional operating systems are monolithic in their design and run most of the drivers and core system services with high privileges (kernel space). A monolithic kernel is easier to build, but comes with the risk that a fault in one of the drivers is able to corrupt the whole system state. Given that the number of bugs increases with software size, the research of the previous decades suggests a more modular design, e. g. a microkernel architecture.

Modern microkernels reduce the amount of kernel responsibilities to a minimum, moving hardware drivers and sometimes even system resource policies to user space. The kernel is responsible for tasks such as low-level address space management, thread management and IPC, while the actual split between kernel and user space implementation depends on the operating system. All components, including drivers, are contained in their own task, benefitting from the hardware isolation provided by a virtual address space. Message passing (called IPC) is used for the communication between the processes.

Due to the increased number of components in a microkernel system, communication between them is critical for the overall system performance. Jochen Liedtke, who researched microkernel design in the 1990s, identified slow message passing to be the root cause of the slow performance of microkernels of the first generation [21]. This is why he declared IPC as "master" in his research and developed the L3 and later the L4 kernel to showcase that microkernels can provide high efficiency with increased security guarantees [21; 22]. The fundamental principle is that of IPC minimality to minimise the time spent in the kernel.

The L4 microkernel has since evolved into a family of operating systems. Examples include SeL4 and L4Re (also known as Fiasco.OC) [7]. The **L4 R**untime **E**nvironment (**L4Re**) is an operating system build on top of a 3rd generation microkernel, called Fiasco. It features a capability-based security architecture, virtualisation of kernel interfaces and more. Almost all traditional kernel functionality runs in user space applications, including paging and I/O handling. This allows the system to be configured to its specific use case [11; 20].

Due to the minimality of IPC in L4Re, communication protocols can grow complex. The programmer needs to take care of many low-level details, such as the order of words, correct bitmasks and more. As a result, many microkernel systems use an Interface Definition Language (IDL) to describe the communication protocols. The advantage of an IDL is its independence of the underlying hardware and the programming language used. Today's L4Re comes with a C++ framework to define IPC interfaces and to share the definitions between the components. It avoids the necessity to learn a separate IDL and helps to abstract from the low-level details of L4 IPC. One of the motivations to move away from an IDL-based design was the argument that most low-level systems software is written in C or C++ and that therefore a tighter integration into C++ is benefitial [8].

With the advent of new systems programming languages that statement is no longer true. This work will focus on Rust, a programming language originally developed at Mozilla for both low-level systems programming and higher-level application development. It is a modern multi-paradigm language with an emphasis on efficiency, safety and concurrency. Its type system allows compile-time memory safety by tracking ownership of objects and references [25], preventing memory management errors such as *use after free* or *double free*. These rules are enforced by a system called the **borrow checker**. Another strength is its ability to help to prevent data races at compile time through a combination of the tracking of ownership and type system features.

Many projects have examined the usefulness of Rust in systems programming [4; 9; 33] and found it to yield comparable performance to C++. To utilise the potential on L4Re, it is required to port Rust to L4Re and to implement an IPC framework able to interface with existing system services. The Rust port to L4Re has been described in [12; 13].

In this work, I will introduce an IPC framework, written entirely in Rust. It will enable L4Re services to be written in Rust. My design was guided by three criteria:

Easy Usability: The interface specification should abstract from the details of an IPC message, such as layout of registers, typed / untyped word transfer, etc. The definition should feel as idiomatic as possible. The user should be able to use the standard Rust compiler (`rustc`) in its stable version. Furthermore, the framework shall enable a developer to use L4Re libraries and services without limitation.

Binary Compatibility: The interface should be able to communicate with other unmodified counterparts, agnostic of whether they were written in Rust or C++.

High Efficiency: IPC is a frequent operation and should not introduce a high overhead in comparison to the C++ implementation, if any.

This work starts with an overview of fundamental concepts of L4Re and its build system as well as Rust and its ecosystem. It will be followed by a chapter outlining adaptation steps necessary for this work, partly carried out in previous work. After a discussion of comparable research projects, the implementation will be described in detail. The evaluation will focus on discussing whether the three design goals were reached by providing performance measurements and discussing advantages and disadvantages of the introduced framework. It ends with an outlook on future work.

2 Fundamentals

This section lays out some of the fundamental concepts required for understanding the introduced Rust framework. It starts with an overview of L4 and L4Re, with a focus on IPC, core libraries and the build system. Last but not least, the Rust language and its ecosystem is introduced.

2.1 L4Re

The L4 Runtime Environment runs on top of the Fiasco.OC microkernel and provides a minimal runtime environment with basic system services and libraries for task, thread, memory, IRQ and I/O management. The Fiasco kernel is a small microkernel providing mechanisms to user space services which implement the system policy on top. Examples for policies implemented in user space include paging, thread control, task management and programming interrupts. An exception is scheduling which has been integrated in the kernel for performance reasons [20].

Each Fiasco task contains a table of kernel objects that it has access to. It can access these through an index into the table called a capability index.[1] A task can only communicate with components of the system for which it has an entry in its table, all other components are invisible. This also applies to hardware resources. Tasks can map capabilities to other tasks to share resource access [20].

In the original design of L4, IPC is synchronous [22]. Liedtke argued that this enables higher performance in comparison to asynchronous message passing such as in Mach. In synchronous IPC, both communication parties, commonly called client and server (or service) are blocked on the IPC call. This allows the kernel to copy the data from one address space to another without an intermediate in-kernel buffer and with only one address space switch.[2] After the message transfer, sender and receiver are unblocked and resume execution. Another efficiency advantage over asynchronous IPC is that the scheduler does not need to be activated between the *send* operation on the client and the *receive* operation on the server. Liedtke observed that clients and

[1] For simplification, the capability index is often also called a *capability*.
[2] Most of the arguments here apply to the case of a single CPU core.

servers repeatedly execute send-and-wait and reply-and-wait operations, respectively. Through the introduction of joint system calls, additional context switches to the kernel are avoided [21].

The IPC primitives in Fiasco work close to the original L4 design [11]. The synchronous communication primitives are augmented with asynchronous notifications, called virtual IRQ [7]. Those, however,are beyond the scope of this work. The original L4 design used global thread handles which served as endpoints for an IPC operation. Fiasco's security architecture is built on top of capability-based access control, avoiding the need for global identifiers. An endpoint in Fiasco is called an IPC gate. Its receiving end is bound to a thread of the server and the sending end is passed to the client, providing a channel for communication. By abstracting from the concrete thread that a message is sent to, the callee is agnostic about its communication partner which can be a user space service or the kernel. A server can listen to multiple IPC gates at once which is why Fiasco allows the registration of a freely chosen label per IPC gate to distinguish different senders of incoming messages.

Thread management happens partly in kernel space and partly in user space. This is why the thread control block is split. The User-Space Thread Control Block (UTCB) contains data required for the management of a thread from user space and resides in a pinned memory page. This includes virtual registers used for transferring and receiving messages. Because the memory page is pinned, the kernel can rely on it being present when copying a message from sender to receiver.

Each message consists of typed and untyped words, the smallest sendable unit. It has the size of a native machine word, e. g. 64 bit on an amd64 machine. Untyped words are normal data with application-specific context and without any meaning to the kernel; they are copied verbatim from source to destination. Typed words are kernel objects (such as capabilities) and explained later on. The virtual registers are divided into buffer and message registers. Message registers are used to send and receive untyped words as well as to *send* typed words. The buffer registers are prepared with flags which instruct the kernel how and where to map the received objects or memory pages to.

Fiasco.OC stores a per-task table of kernel objects (also known as capabilities) to which a task has access. The user application receives a capability index (similar to a handle) into this table and uses it for all operations with the kernel object. An example is the IPC gate which is accessed through a capability to it. As explained above, the IPC gate abstraction leaves the user ignorant of the participant on the other side which is in fact true for all capabilities. The type of capability is therefore not defined by its access method, but just by the communication protocol being used. Since capabilities are stored in a per-task table and since operations on capabilities can only be invoked through the index into this table, a task is unable to sneak a new index and escalate its privileges. Since all resources are accessed using capabilities,

including fundamental things like memory, the system developer can control how the task will see its surrounding.

Capabilities and memory pages are sent using the Flexpage abstraction. A Flexpage is called a typed word and describes access rights, the number of objects or pages to transmit and the operation. For this work, the most important type is the object Flexpage, used to transmit capabilities. The send operation of a Flexpage is either map or grant. If it is mapped, both the receiver and the sender can access the object or memory page. The sender may revoke the mapping at any time. The grant operation on the other hand exclusively transfers the Flexpage to the destination task and the sender will lose its mapping [11; 17; 21].

Before a Flexpage can be mapped or granted, the receiver must prepare for it and instruct the kernel where to map the received page to with corresponding flags in the buffer registers. For memory Flexpages, this is the location of the memory to map the page to, for object Flexpages this is the capability index in the task-local table. The specified destination is also referred to as “receive window” or “receive slot”.

L4Re messages are described by a message tag which is an additional tag attached to each message (including replies) being sent. It has the size of one machine word and contains information about the used protocol, the number of untyped and typed words to transmit and additional flags. It plays a central role in the communication, because it is important both for the kernel and the receiver. The receiver uses the protocol information to discriminate between protocols and uses the word and item count to verify the length of the message. The kernel uses word and item count to only copy the relevant bytes of the registers. In a reply, the protocol field is “free” to use and is therefore used for status indications by convention. Negative values indicate a failure and 0 indicates success. Positive numbers are free to be assigned by the protocol.

Most modern microkernel-based systems use an IDL to define the protocol interactions [8; 9] and to hide IPC complexity. An IDL is a custom language that allows an abstract definition of the client-server relation. The IDL-compiler uses the specification to generate client and server stubs. Adding support for a new programming language is as easy as to add a new language exporter. It can be argued that the downsides outweigh the upsides: an out-of-source definition requires additional mental maintenance overhead and requires the programmer to learn an additional language [8]. A few years ago, low-level services were written predominantly in C or C++ so that the language interoperability was not as important [8]. This led to the development of a C++ framework that used streams to abstract from the serialisation details. Disadvantages of this approach are that it requires the programmer to write data in the correct order and read it the same way on the receiving side. It also prevented sharing of the IPC code between client and server implementations. It was therefore replaced by a new C++ framework with a declarative approach. A protocol is rep-

resented as a C++ class, with methods representing the operations that the protocol supports. For the client, calling a remote functionality on a protocol-typed capability is the same as calling a "normal" class member function. Specialised data types abstract from the low-level details and serialise to a custom binary transmission format, tailored towards efficiency. The class can be shared among client and server, allowing code reuse and easier updates of the Application Programming Interface (API). The order of typed and untyped words is transparently handled and the developer can choose any argument order in the interface methods.

2.2 Rust

Rust is a systems programming language with an emphasis on safety, concurrency and speed. It features a modern type system, providing memory safety guarantees previously known from garbage collected languages. This chapter gives an overview of the most important Rust concepts which required to follow code examples and implementation paradigms throughout the chapters. It is followed by an overview about the Rust ecosystem, including the build and project manager `Cargo`.

2.2.1 Language Basics

Rust is suited for systems programming because of the possibility to build efficient abstractions of low-level details while retaining control over memory layout and allocation. Example research and volunteer projects demonstrate the applicability of Rust for programming low-level user space and kernel functionality.[3]

One key aspect of Rust's success is its ownership model, where each object has exactly one owner at a time. The compiler tracks the lifetime of each object and if it goes out of scope, the compiler inserts the required cleanup code. The compiler annotates each object and each reference with an implicit lifetime to track its existence. The programmer can aid the compiler by adding explicit lifetime annotations. Lifetimes can also depend on other lifetimes, so that an object is valid as long as its dependent object exists. This is enforced by three rules:

1. Each value in Rust has a variable that's called its *owner*.
2. There can only be one owner at a time.

[3] Further information on this can be found in the related work.

3. When the owner goes out of scope, the value is dropped (i. e. clean up actions such as free).

In addition, objects can be referenced any number of times immutably, as long as no mutable reference exists. Mutable references have to be unique. With these *borrowing* rules, the compiler can reason statically when an object behind a variable ceases to be used. For standard types, the compiler knows how to *drop* (free) the object, but the programmer can extend this for custom data types. The memory management policy of the ownership model provides safety guarantees previously known from garbage-collected languages.

To abstract from concrete implementations, Rust offers **traits**, describing functions, types and constants required for a type. They are similar to interfaces in other languages (such as Java). In addition to defining functionality, traits are also used to mark a certain property of a type without an implementation. This can be used in generic functions or data types to enforce a property of an argument. It thereby serves a documentary purpose as well. An example is the built-in `Send` trait. Types implementing it need to make sure that it is safe to move the type across thread boundaries.[4] Traits can be marked as unsafe, alerting the programmer to pay special attention when implementing the trait. `Send` is marked as such, so the programmer must make sure that the data type can be copied across thread boundaries and is valid without further state. After the send operation, the sender has lost access to the object, while the receiver has gained ownership.

The `unsafe` keyword in Rust allows the programmer to bypass some of Rust's safety rules in a block or function. Using an unsafe block gives the programmer the ability to dereference raw pointers, call functions marked as unsafe,[5] implement unsafe traits, mutate global state or access union fields [35]. It is apparent that this keeps most other safety rules untouched. The majority of tasks can be performed without unsafe Rust [35] and in fact only a small fraction of the standard library (*std*) relies on it. An example are the Foreign Function Interface (FFI) bindings to interface with existing (C) libraries.[6] Marking a few lines of code as unsafe offers the benefit that in the case of an invalid (memory) operation, the error can be traced within the unsafe blocks and not in the safe abstractions in the levels above [16].

Abstractions over unsafe code need to deal with memory alignment or with a logical association between types where no memory reference exists. This can be addressed by a feature of Rust called "zero-sized types" which do not require any space in

[4] `Send` is auto-derived for primitive types by the compiler and only needs to be implemented manually for complex data types.

[5] This also includes functions from non-Rust libraries which are inherently unsafe to call.

[6] There are more examples such as doubly linked lists or certain performance optimisations only expressible through pointer arithmetic. The inclined reader may want to read [16, chap. 19.1] and the Rustonomicon [35].

memory and their load/store instructions can be optimised away. This is for instance useful for a generic member of a data struct. If the struct is parameterised with a zero-sized type, it will not consume any space within the parent struct and calls to methods are optimised to function calls. The Rustonomicon contains a more detailed overview [35]. One commonly used zero-sized type is the `PhantomData`. It can be parameterised over arbitrary generics and helps to avoid errors due to unused generics, for cases where the type dependency is implicit.

To ease the understanding of code examples in the subsequent chapters, it is useful to understand the concept of error handling in Rust. Rust leverages the type system to handle errors and enforces explicit error handling whenever an error can occur. Functions that can produce erroneous results return the enumeration type `Result<V, E>`, where `V` is the value type and `E` the error type. Enums in Rust are like tagged unions and can only contain one of the possible states, here `Ok` or `Err`. This forces the programmer to reason about possible failures when consuming the data. For shortening examples and to make the understanding of provided examples as straightforward as possible, all examples will use the `unwrap()` function, defined for the `Result<T, E>` type. It returns the value if present and aborts the program with a message (called a panic) otherwise.

Safe Rust does not permit uninitialised variables or dangling pointers. If a value may be present or not, the `Option<T>` offers the programmer to choose between `Some(value)` and `None`. The usage of this pattern within the core language makes null pointer and uninitialised values superfluous.[7] Similar to the `Result` type, `Option` enforces explicit handling of the absence of a value and documents this in the type signature.

Whenever a method on an object is invoked, the compiler inserts a call to the associated function and passes a reference to the object as the first argument. This is called static dispatch. In case of a generic function, the compiler has to know the target type to invoke the method on the corresponding target type. Rust favours static dispatch, performing monomorphisation during compile time.[8] This is feasible since Rust does not offer object inheritance and hence does not deal with overriding. In certain situations, dynamic dispatch is more appropriate, especially if the object passed cannot be determined during compile time. An example would be the usage of a vector with different types, all implementing a certain trait. The developer needs to explicitly opt into dynamic dispatch by casting an object reference into a trait object. The type information of the original type is lost and only the methods from the underlying trait are callable [37]. A trait object is a special reference and is referred to as fat pointer. This is because trait objects are in fact a reference to the target object and a

[7]The standard library provides helpers to create dangling pointers and uninitialised memory for cases where this is required.

[8]Generic types are specialised during compile time and function references resolved. This avoids the need to track type information during runtime.

pointer to the virtual method table (vtable). This is different to C++ where the vtable is stored with the class definition and consulted every time a "virtual" function is called. This optimisation allows a method of a struct to be statically dispatched in the general case and dispatched dynamically if requested. This has the drawback that a trait object is larger than a normal pointer and cannot be used for FFI operations [2; 16, chap. 17.2].

2.2.2 Ecosystem and Build Tools

The Rust ecosystem is made accessible using a set of tools, all supported by the different Rust teams. The components work seamlessly together, but are not always compatible to existing solutions due to the nature of Rust. This section shall focus on the aspects and tools which I have used for the framework development.

Libraries The first challenge lies within an incompatible library format used by the Rust compiler. The advanced type system features of Rust require a revised Application Binary Interface (ABI), incompatible to the one of C. C libraries ship the compiled code and the definitions separated in archive files (`.a`) and header files (`.h`). In C++, advanced features such as templates can only be used in header files because there is no way of representing the types in the binary. Rust bundles compiled code and type meta information in a custom library format with the file extension `.rlib`. This avoids additional header files, but is incompatible to the workflow of most build systems. The resulting adaption challenges are discussed in Section 4.1.1. To differentiate from non-Rust libraries, the term crate has been coined in the Rust community, but is used with the term library interchangeably. In subsequent chapters, the term crate is used to emphasize that the library is written in Rust.

`Cargo` `Cargo` is the project manager and build tool for Rust projects. Each Rust project contains a `Cargo.toml` configuration file, specifying dependencies, crate features,[9] build scripts and more. Build scripts are small Rust programs compiled before the actual library or program and perform the setup of the build environment. This may be the preparation of a C dependency library or code generation for the library. For instance, C library bindings can be auto-derived from header files using a library called *Bindgen*. Bindgen is invoked from the build script and the generated C ABI definitions are then included by the crate.
`Cargo` also automatically fetches dependencies from the internet, configures them and resolves version conflicts. It is also used to execute tests.

[9]Features is the Rust term for conditional compilation.

Rust Compiler Versions The Rust compiler comes in three different versions: the **nightly** (development) compiler, the **beta** testing version and **stable**, meant for production use. Nightly Rust contains many features not present in stable. Some of them have not been stabilised, others will disappear again. This is why I decided to make use of the stable Rust version. Previous work suggested that stable Rust lacks certain functionality to be ready for systems programming [4; 9]. This work will discuss the situation for this project in 2019.

2.2.3 Macros

To avoid repetition or to simplify an existing work flow, most languages offer macros as a method to transform code from a simpler into a more complex form. C and C++ use a simple search-and-replace macro system, where occurrences of terms are replaced by the defined macro body. Rust offers two macro systems which are used in the Rust IPC framework and which are introduced in this chapter.

The simpler system, sometimes referred to as "1.0 macros" or "`macro_rules`" transform from and into a Rust Abstract Syntax Tree (AST). During definition, the programmer can match on specific tokens such as identifiers, types, expressions, blocks, literals, etc. and generate new code with the captured inputs. As soon as a macro call does not match the declared (syntactic) input parameters, the compiler will assert the exact position within the macro call and warn the user which syntax element was expected. An example from the standard library is the `vec!` macro which allows the creation of a vector initialised with the specified elements. The expression `vec![1,2,3]` is transformed into vector initialisation code and into push operations. Another example is the `println!` macro for displaying text on the console. It mimics a function with a variable number of arguments, which is not supported in Rust.

Rust macros are hygienic in the sense that variables defined within a macro do not leak the macro evaluation body and that they can only access input parameters, but not state from outside the macro body [15; 16]. In comparison to their C counterparts, they are safe to use due to their ability to distinguish between different input syntax elements and due to their hygiene.

Since the Rust 2018 edition, Rust supports another kind of macros, called procedural macros (proc macros for short). These are like procedures run during compile-time and can be pictured as compiler plugins for code transformation. In contrast to the higher-level `macro_rules`, they operate on the token level, offering a greater degree of flexibility, but require manual parsing work of the token stream. To aid the macro author, crates such as *syn* and *quote* exist which can be used to parse the Rust source code, transform the syntax elements and to serialise the data structures into a token

stream again. This way, the developer can write its macro logic in Rust and benefits from the already provided parser implementation.

There are three different proc macro types: custom derives, attribute-like macros and function-like macros, shown in Listing 2.1. Custom derives are annotations that allow the automated implementation of traits during compile-time for data structs, enums or unions. Listing 2.1 shows this on lines 1–2 for the built-in `Clone` trait.

Lines 4–5 show an example from the rocket web framework. The function is transformed using the custom `get` macro, using its arguments and the function body below. The macro inserts code to parse the HTTP requests in a memory-safe way. This includes automatically checking the type of age and the validity of the `name` string.

Listing 2.1: Procedural Macro Examples

```
#[derive(Clone)]
struct Money(f64);

#[get("/hello/<name>/<age>")]
fn hello(name: String, age: u8) -> String { }

fn error(reason: &str) -> String {
  html!(<html><body>
    <p>#reason</p></p></body></html>
}
```

Lines 7–10 show a non-implemented HTML macro, generating an error page from a string, including a reason. This is also a demonstration on the powerful capabilities of procedural macros.
The macro author is able to fine-control each transformation step of a procedural macro. If an error occurs, the macro execution can be aborted and the exact location along with a custom error message can be reported to the user. This means that even for Domain-Specific Languages (DSLs) like the HTML example above, meaningful error messages can be reported if the macro is used incorrectly.

3 Related Work

Communication between threads, processes or machines is an essential part of concurrent and distributed programming. Conceptionally, messages are serialised and then copied between the entities. The transport mechanisms vary considerably. While microkernels offer kernel-guarded mechanisms to copy data privately, Remote Procedure Call (RPC) mechanisms are used commonly over lossy and potentially public network connections. This leads to different design decisions for data serialisation, protocol setup, etc. The most generic communication method is RPC, with which this chapter starts. It moves on to a generic view on IPC on microkernel systems which feature a Rust port. A case study for IPC in Go is shown afterwards and the chapter closes with a discussion of the different approaches.

3.1 Rust RPC Libraries

A remote procedure call mimics function call behaviour for contacting a remote process. It transparently serialises data, contacts the remote services and deserialises the reply. Due to Rust's frequent application in network programming, this is often communication over a network connection, as can be seen below. The main motivation is to ease the development of client-server architectures.

Servo IPC channels Rust provides a communication abstraction in the standard library, called a channel. It allows exchanging messages between threads in a unidirectional fashion. Since it is implemented using shared memory, this work only within the same process. The *ipc-channel* is a drop-in API replacement for the *std* version, using different communication mechanisms, depending on the platform, to realise communication between threads of different processes. The used channel implementation can be swapped easily by changing the type import.

Cap'n Proto This library allows exchanging data in a machine architecture and programming language independent format. It offers in-memory serialisation and deserialisation of data without copying [3]. The communication protocol is described

using a custom DSL from which stubs for working with the interface are generated. The system is easily extensible to new programming languages by extending its compiler with a new language exporter.

Tokio-rpc This crate is part of the *Tokio* framework, implementing a runtime for efficient asynchronous I/O programming, including abstractions for asynchronous network communication and work-stealing task schedulers for locally distributed computing. The RPC module of this framework helps to distribute the asynchronous workloads over a network [1]. Messages are serialised into network requests and the replies are processed asynchronously using the Tokio event loop.

TARPC Tim and Adams RPC library is a stand-alone crate providing serialisation and deserialisation of function parameters and return values for RPC. According to the project, its main focus is ease of use. The definition of the remote procedures happens in a trait-inspired syntax within a macro. The macro knows the arguments for each remote procedure and can therefore derive certain parts of the client and the server automatically. It uses a network as its transport layer.

3.2 Rust on Other Microkernels

Most microkernel systems use an IDL to define the IPC communication, examples being Barrelfish, old versions of L4Re, Fuchsia and SeL4 [6; 8; 9; 31]. An exception is Redox OS, a UNIX-like microkernel operating system, written entirely in Rust. It lacks documentation on an IDL or an IPC framework and will therefore not be discussed here [33].

Barrelfish Barrelfish has been developed to research new ways of working with multi- and many-core systems. Each core is modelled as a separate machine, communicating solely through message passing. The system is based on a microkernel with the attempt to move as much functionality into user space as possible. This includes memory allocation as well as thread and task scheduling [9].

Porting Rust to Barrelfish required a low-level *libbarrelfish* to be written, exposing low-level functions to be used by the *std* port. Adaptations to *std* included a custom `channel` implementation, since threads can migrate across cores and hence share not necessarily the same address space [9]. Inter-process communication in Barrelfish is split into a core-local and a cross-core-core case. Messages on a single core are passed

using CPU registers. For core-core communication, a shared memory region is used. To ease the interaction between services and to help to abstract from the complexity of communication protocols, an IDL called Flounder is used. It generates the communication stubs in the requested programming language. The Flounder compiler was extended to generate Rust function and type stubs. The stubs for languages other than Rust make use of the core system library called `libbarrelfish` and the Rust stubs of the crate *liblibbarrelfish*, respectively. In contrast to systems of the L4 family, sending and receiving is non-blocking by default in Barrelfish and uses callbacks when a message arrives. The data needs to be freed from the internal message buffer only after the send and receive operations have been completed. This is a non-static lifetime impossible to express in Rust: the object lives as long as the IPC lasts and therefore `rustc` cannot reason about the deallocation statically. To resolve this, data is copied out of the Rust data type into the buffer at the sender and from the receive buffer into a newly allocated Rust object on the other end. This way, Rust can again take care of managing the memory objects without affecting the sending process. The code for this is generated by Flounder. Rust gains access to all system services by accessing the generated Flounder stubs.

SeL4 SeL4 is a microkernel from the L4 family with synchronous IPC and a focus on security, including the usage of object capabilities. Furthermore, asynchronous notifications extend the synchronous IPC, roughly akin to L4Re's virtual IRQ [7]. Service interfaces are defined using an IDL compiler. They connect endpoints which are bound to a specific thread and incoming messages are disambiguated using badges (akin to L4Re labels).
Aside from the existing SeL4 infrastructure, there is a project called Robigalia, aiming to replace the entire userland by software written in Rust [29; 30]. The project is still in an early stage and the implementation is not complete. Compatibility is not a concern because all components of the existing SeL4 userland are rewritten. This also includes abstractions of which some are not fully formulated yet [29; 30]. There is no documentation on the usage of an IDL.

Fuchsia Fuchsia is a new microkernel operating system developed by Google. Design principles include capability-based resource access and control, local namespaces (allowing virtualisation of resources) and a minimal kernel design. IPC is synchronous for the client, but asynchronous for the server [6; 10]. Fuchsia has support for Rust clients and servers through its IDL compiler called *FIDL*.

3.3 L4Re IPC in Other Languages

The recommended way of handling IPC on L4Re is the C++ framework. It abstracts from many low-level details, such as the message tag, buffer register setup for Flexpage mappings, etc. In the C version, these details need to be implemented for each call separately. The details are wrapped in library functions. The L4Re snapshots ship support for Python, Fortran and OCAML. In these languages, no access to the C++ framework exists. All IPC interfaces are accessed using the C wrapper libraries.

Go is a modern programming language developed by Google to provide an alternative to C and C++ in high-level systems programming. It achieves memory safety through garbage collection and comes with a modern type system. Through mechanisms like Goroutines and channels built directly into the language, concurrent programming is made convenient and is encouraged. Goroutines are the Go term for user-level threads which get mapped to OS threads by the Go runtime. Channels are a mechanism to exchange data bidirectionally among Goroutines without the need to explicitly share data, thus avoiding data races [26].

The aim of the Go port for L4Re was not optimal performance, but the best possible integration of the Go channels with L4Re IPC, while keeping the changes to the Go runtime minimal.

Go comes with a runtime containing the garbage collector and a goroutine scheduler. Goroutines are much faster than kernel threads, since they are set up and scheduled in user space, avoiding a switch to kernel space. But since the kernel is agnostic of the Goroutines, a call will block the thread from the Goroutine thread pool which did the call. The Goroutine scheduler will then migrate the Goroutine to a different thread from the pool, continuing execution, instead of waiting for the reply. The solution to this dilemma is discussed in [26] and a comparison with Rust is done in [12].

Go channels are an important language design decision to enable the construction of highly concurrent programs by encouraging the developer to use message passing instead of state sharing. Channels are tightly integrated into the language, for instance, a special operator exists to read and write from and into channels. The work Aimed to use Go channels for IPC due to its wide-spread use in the Go ecosystem [26]. Since Go lacks operator overloading, an extension can only be made by patching the compiler or the runtime, while [26] decided for the latter. To couple channels to IPC operations, the runtime was extended to copy messages from the channel buffer to the UTCB and vice versa transparently. The duplex connectivity expected by a Go channel is provided by two IPC connections opened during the initialisation.

An essential part of L4Re is the transfer of capabilities using IPC, to enable a dynamic interaction of services. The transfer of capabilities is similar to that of data, except that

they are not stored in the message registers, but in the buffer registers of the UTCB. Similar to sending a message synchronously, sending a capability also requires that the other party expects it. A capability slot needs to be preallocated on the receiver side. Therefore, specialised methods for receiving and sending capabilities were introduced to Go. The new receiver methods take care of allocating a capability slot before the receiving thread awaits the arrival of the capability.

3.4 Discussion

This chapter gave an overview about two classes of message passing: RPC-style communication tailored for message exchange independent from the platform and IDL-based IPC, tailored to allow communication on a specific operating system with high performance. Microkernel systems using "raw" IPC without any IDL or framework are not discussed here because the lack of abstraction from the low-level IPC details make them incomparable to this work. This especially applies to the Robigalia and the Redox project.

3.4.1 Remote Procedure Call Libraries

RPC libraries are built to communicate with remote services, commonly over a network. A network connection is lossy and protocols need to be able to handle connection failures. The overhead incurred by these checks is too high for IPC, because context switches are frequent in microkernels and overhead is not tolerable. Furthermore, communication between entities on a microkernel system is not lossy and private, hence design decisions are not applicable to microkernel IPC.

This particularly applies to *TARPC*. While the service definition looks clean and easy, its implementation is bound to a network connection. By using *bincode* for serialisation, it is incompatible to the serialisation used by the L4Re C++ framework. Due to the custom syntax within the *TARPC* macro, the programmer needs to learn its usage and IDEs are unable to assist with syntax highlighting.

The Tokio framework has similar issues as TARPC and Servo IPC channels. Additionally, its asynchronous nature makes it unsuitable for L4Re IPC which is synchronous. Tokio uses a large event loop to dispatch work to threads or processes which communicate using the *tokio-rpc* library. The communication happens via different transport methods such as UNIX sockets or network connections. The provided communication abstraction incurs too much overhead. Furthermore, messages are serialised using `Protobuf`, a serialisation format incompatible to that of the L4Re framework.

An exception is Cap'n Proto which has been designed to provide efficient protocol specifications. Its serialisation capabilities are designed to be independent of the underlying transport mechanism and are focussed on efficiency. Its usage of a DSL requires the programmer to maintain an out-of-source interface definition, but eases the interoperability between programming languages. The usefulness for L4Re would need to be evaluated separately, but at the moment, this would mean reintroducing a less specific IDL to L4Re again. It also has the downside to use a binary format incompatible to the existing L4Re framework which is also not exchangeable.

Rust offers the possibility to define custom DSLs which can be tailored to the use case. This was used to provide compile-time safety for network protocol specifications [4]. Packages are defined using an ordinary struct which is annotated with a custom attribute to mark it as a package. The language allows custom type fields such as `u16be`, a `u16` in big endian and adds additional annotations on each struct member to specify additional properties. These modifications are called a "compiler plugin". In the retrospective, [4] used an early version of procedural macros which have been stabilised in Rust 2018. It demonstrates the power of statically checking properties of data structures and their interactions, but also names a few deficiencies. The most prevalent one is the inability of procedural macros to type-check the expressions that they operate on. The DSL from the thesis is tailored to network processing and thus cannot be used for designing IPC interfaces, but it shows how to use Rust's meta programming techniques for domain-specific language extension.

3.4.2 IDL-based interface definition

IDLs are the first choice for most microkernel operating systems when trying to simplify the handling of IPC [9; 10; 31]. An IDL offers the great benefit that it is a common language used throughout the system and that its specification is independent of the applications programming language. Adding support for a new programming language to an IDL compiler comes down to adding new code generators which will give access to all other system services. L4Re IPC evolved from an IDL-based solution to a C++ framework for message passing [8]. Reintroducing a new IDL to L4Re is hence counterproductive. It would require the IDL to be able to export not only to Rust, but also to the two "official" languages, C and C++. Service authors of non-Rust applications would need to maintain both the IDL-based interface and the one provided by C++.

The two promising projects Robigalia and Redox do not use any IPC abstraction. Redox has the aim to be a UNIX operating system built on top of a microkernel. IPC has not been a focus of the project yet and calls are still implemented manually [33]. This also applies to Robigalia. Since the full userland of SeL4 shall be replaced, no

compatibility concerns exist. The IDL of the SeL4 project is not used. Due to the early design phase of Robigalia, no inspiration can be drawn from its IPC approach.

3.4.3 L4Re IPC Interfaces In Other Programming Languages

As pointed out earlier, most programming languages use the C ABI to interface with system services or to implement a service. Using this interface, the IPC calls will not feel idiomatic, no matter the target language and potentially require idiomatic glue code to be written for each interface. Languages other than C++ can call C functions only using the defined C ABI. Functions need to exist as public symbols in the resulting binary for the compiler (or interpreter) to generate a function call to it. Given the large number of small C functions involved for the IPC call setup, this is a considerable performance penalty, since these function calls are all inlined in C or C++. It is therefore something I wanted to avoid in the Rust version.

The aim of the port of Go to L4Re was an idiomatic integration using the language-provided channels. Since many libraries use the built-in channels, this would allow a greater reuse of software. Before adapting the language to enhance Go's channel implementation, the author investigated the reuse of existing RPC libraries for Go, in this case the *netchan* library. In contrast to Go channels, *netchan* implements channel as objects with send and receive methods. It cannot make use of the dedicated channel operator syntax because of the lack of operator overloading. Using this library gives the advantage that the RPC interface is consistent across platforms. The downside is its mandatory usage of a network connection which is a high-level abstraction with much overhead for communication, contradicting the principle of minimality of IPC on L4Re. Network-based message passing has different design requirements in terms of data integrity, performance and serialisation. This approach also makes it impossible to send capabilities which is why the author did not consider the extension of this library. This is similar to the discussion in chapter 3.1, where similar approaches for Rust are shown.

For the Go port, the thesis author decided to adapt the Go runtime to integrate the L4Re IPC mechanisms into the core language. The resulting API usage is comparable to the C++-stream-based API of L4Re, the predecessor of the current IPC framework [8]. Many problems presented in [26] are not applicable to Rust and hence do not help in defining an idiomatic IPC framework. Rust's channels are part of the standard library and can hence be replaced. It also builds on a much smaller runtime, comparable to that one of C++ [16].

4 L4Re and Rust Infrastructure Adaptation

The L4 Runtime Environment offers great flexibility to configure the system in low-level details such as memory allocation or whether file system access is provided. The system is highly modularised, meaning that there is no common set of functionality that can be expected on every system. The core components such as the common API to system services, including IPC, is implemented in the L4(Re) core libraries, located in a directory called `l4re-core`. For this work, the `l4sys` and `l4re` are the most important ones. The directory also contains core services such as Moe and Ned.[1]

The `l4sys` directory contains the basic C (and C++) implementation for accessing the UTCB, doing system calls, sending Flexpages and more. It also contains the majority of the C++ IPC framework. Because it does not make use of the `libc`, this library can be used in any application.[2]

The `l4re` and `l4re_c` directories contain user space abstractions provided by L4Re, along with a default implementation for most of them. Examples include the data-space, region manager or the memalloc interface. These build on the low-level IPC mechanisms of the `l4sys` library and allow the creation of dynamic programs with memory allocation, etc. `l4re_c` acts as a thin layer of glue code to allow the usage of the C++ interfaces from C programs.

L4Re comes with a C standard library called Uclibc. It translates most UNIX functions into L4Re IPC calls to the corresponding servers. This aids to port programs using UNIX/POSIX interfaces. The version of Uclibc shipped with L4Re has been adapted to use L4 IPC to IPC services instead of UNIX syscalls.[3] It has been split into multiple libraries, each implementing a specific subset of a functionality, for instance signal and memory handling.

[1] Moe is the root task of L4Re providing basic services. It starts Ned which is the init process which bootstraps the system. See L4Re's documentation on `https://l4re.org/doc/`.

[2] In fact, the adapted C library Uclibc is built on top of multiple backend libraries using the `l4sys` package.

[3] This does not imply that Uclibc is POSIX-compatible, but it provides a UNIX-alike API for most use cases.

Before diving into the details of L4Re specifics, it has to be noted that Rust's standard library *std* is already ported to L4Re [12]. It assumes a certain set of services to be present in the system and accesses it through the `libc`. It builds on nearly a dozen libraries such as `core` or `alloc` and acts as a common façade for it; applications can be built with only a few of them if *std* is not appropriate. Additionally, *std* also implements many high-level APIs, for instance for threading, message passing, date and time handling and more. As described in [12; 13], I patched *std* so that it uses Uclibc on L4Re. Since the Uclibc interface is similar to that of other C libraries, the small amount of changes to *std* was accepted upstream.

4.1 Build System Adaptation

L4Re is built using a recursive (GNU) Make-based build system called BID. Each directory contains a *Makefile*, declaring the directory either as a library, a program, header include or a subdirectory of the source tree.[4] I will focus on the modifications to the library rules (in `lib.mk`) and program rules (`prog.mk`). Rust does not use include files and hence the `include.mk` is left untouched. The L4 source tree is structured into packages. Each Makefile defines a variable `PKGDIR`, specifying the path to the package root that it belongs to.

Each BID package needs to define input files, output targets and dependency requirements. An example is given in the next section. This information overlaps with that present in the `Cargo.toml` file (see Section 2.2.2 on page 14). In my previous work [12; 13], I decided to only integrate the compiler `rustc`, similar to the already supported ones (GCC / Fortran). This proved to be difficult, since the Rust ecosystem expects the Rust compiler to be coordinated by `Cargo`. The compiler also carries out linking to allow for cross-crate optimisations and link-time optimisations and hence does not permit calling the linker separately. Another reason for this requirement is that libraries and build scripts may add additional libraries and linker arguments and the compiler makes sure that these are integrated into the linker invocation. This has been traditionally the task of BID. Features such as conditional compilation and build scripts were impossible with this solution.

Another selling point of Rust is its central crate registry (crates.io), containing a large number of free and open source libraries. An example is the *Bitflags* library, making bit flag manipulations less error-prone and type-safe without overhead.

In order to keep the L4Re adaptations close to the upstream Rust ecosystem, I decided to let BID handle all L4Re-related packages, while `Cargo` manages all Rust-related crates. In this scenario, BID invokes `Cargo` whenever it detects Rust source files and

[4] There are a few more possibilities which are not relevant for most use cases.

handles its output files afterwards. Cargo in turn orchestrates rustc which then takes care of calling the linker. This avoids conflicts between the two build systems because Cargo looks like an ordinary compiler to BID.

It is possible to pass arguments to rustc through environment variables that are interpreted by Cargo[5] [34]. BID can use these variables to directly influence the compilation process. With the environment variables set, Cargo can compile libraries and applications. It fetches all dependencies from the crates.io registry, executes build scripts and continues with the actual compilation.

Throughout the next sections, I will explain the chosen boundary between both dependency managers in more detail. The next section will start with BID-based crates and how they differ in contrast to a non-Rust L4Re library package. It is followed by a section on application creation and linking.

4.1.1 Libraries

Rust libraries store both compiled binary objects and abstract type information about generic types. In contrast to languages such as C/C++, this avoids the need for additional include files. This file format has the extension .rlib and is often referred to with the same name. The normal work flow for BID is to translate source files to object files and call the archiver to produce a static library. rustc handles the whole process transparently. I extended the library make rules to detect Rust source files and call rustc in one step. Afterwards, BID resumes with the PC file generation. PC files are files used by pkgconfig to aid linking. Even though BID does not use pkgconfig directly, it uses its format to store linker information for each library. Rust libraries use different mechanisms to pass linker arguments, rendering PC files useless. For compatibility reasons, PC files are kept and instead of linker arguments, the path to the rlib is passed. This will pass the library as an argument to the linker. This works because the rlib format is based on the archive format (.a) used by C/C++. This is an undocumented implementation detail and not guaranteed to work in the future. Rustc passes rlib's to the linker in the same way and hence I decided to copy this behaviour. The details of the Rust library format and their usage by the Rust compiler are not relevant for library development on L4Re because they are hidden by BID and Cargo.

Listing 4.1 shows a Makefile with a look akin to other L4Re Makefiles. It defines a package *l4-sys*, explained in more detail in Section 4.2 on page 29.

[5] An example is CARGO_BUILD_RUSTFLAGS.

Listing 4.1: Makefile for the *l4-sys-rust* library

```
1  PKGDIR ?= .
2  L4DIR ?= $(PKGDIR)/../../..
3
4  TARGET = libl4_sys-rust.rlib
5  SRC_RS = lib.rs
6
7  REQUIRES_LIBS = l4sys libl4re-wrapper
8  include $(L4DIR)/mk/lib.mk
```

The package has a simple layout and builds only one target. In the first line, the package directory is declared. Relative to it the directory with the basic L4 files on line 2. Line 4 describes the name and output type of the library. Originally, the rules in `lib.mk` expect a library name to end on `.a` or `.so`, for static and dynamic libraries, respectively. Rust links statically by default and therefore only the static library format `rlib` is considered here. Even though the library's name is *l4-sys*, the file name ends on `-rust`. `Cargo` and `rustc` use a file name suffix with a hash to discriminate different versions of the same library to allow a dependency to occur multiple times in a dependency graph. Since this library is built by BID, the hash is replaced by the `-rust` suffix and ignored otherwise. This way the compiler can still split off the end of the file name without destroying the library name. To summarise, an L4 Rust library target line must contain a file name starting with `lib` and end on `-rust.rlib`. If the library name contains hyphens, they need to be replaced by underscores.

On line 5 of Listing 4.1, the Makefile continues with the specification of the source files. In contrast to C, the compiler requires only the reference to the main source file and can find all other source files by traversing the module declarations within the files recursively. I have added a new variable called `SRC_RS`, specifying where to find the Rust sources. Listing 4.1 closes with declarations of the dependencies and an include directive, specifying the type of output.

In addition to the variables shown in Listing 4.1, the package author can extend the flags passed to the Rust compiler by specifying `RS_FLAGS`. This is not required for most packages, since BID will make sure that all relevant arguments are chosen appropriately.

4.1.2 Applications

BID builds applications in two steps: it translates the source files to compiled objects and calls the linker afterwards. The separation is necessary, since an L4Re binary is arranged differently in comparison to a Linux binary, for instance. It requires custom

startup and teardown code in the binary and the used ELF sections also differ. Such details are hidden for most platforms since the linker knows the target platforms. Hence, a cross-compilation tool chain comes with all required files. L4Re's linker script as well as the custom startup/teardown code is not published upstream and thus needs to be passed to each linker invocation. The rest of its command-line arguments is assembled from its knowledge of the library dependencies.

In my first solution [13], I decided to compile applications into static libraries with a C-ABI-compatible main function. The archive was then passed to `ld` which detected the function as entry point and linked a binary. This worked because BID called `rustc` directly and instructed it to emit a static library instead of an application. This solution leaves the linking process with BID, hence getting rid of the issues involved with it. The downside of this approach is that it loses the convenience offered by `Cargo`, such as the download and management of crates, the execution of build scripts and more. It also forces each application to declare the main function as `extern "C"` for the linker to find the entry symbol.

With the experience from the previous integration attempt, I decided to integrate `Cargo` instead of `rustc` into BID. I decided to join the compilation and linking step into a single one, in which BID would generate the linker arguments that it needs to pass. The generated argument list is passed using the `CARGO_BUILD_RUSTFLAGS` environment variable. Cargo passes these flags on to `rustc` which integrates them into its linker argument list.

To extend Rust's portability to other platforms, it comes with a built-in list of target linkers that it supports. It uses the GCC linker `ld` on most platforms, but supports other linkers such as Gold. Recent snapshots of L4Re have introduced a script called `l4-bender` which is used by BID for linking. It wraps the linker call and abstracts from the cross-platform linking details. It uses PC files (as used by *pkgconfig*) to retrieve the linker flags required for linking libraries. BID passes a list of libraries to `l4-bender` which then loads the corresponding PC files and carries out the linking. This is a benefit for `rustc` because this means that the number of arguments which need to be passed to the linker decreases and with it the complexity.

The call syntax of `l4-bender` is different to that of `ld`, thus the Rust compiler cannot work with it by default. I added a new linker specification to the Rust compiler, supporting the invocation syntax of `l4-bender`. The target specification for L4Re has been adapted to use this linker variant by default. However, `l4-bender` introduces a new problem: some of the arguments to it contain spaces. In the `CARGO_BUILD_RUSTFLAGS` environment variable, spaces are used as a delimiter between the command-line arguments. Including arguments with spaces in the environment variable is hence impossible. At the time of writing, this issue has been reported, but not addressed yet. As a temporary solution, I introduced another environment variable `L4_BENDER_ARGS`; it is parsed by `rustc` and allows to escape spaces in arguments using quotes (similar

to the UNIX shell syntax). This solution has a temporary character and was hence rejected upstream.

4.2 L4rust Libraries

The aim of competitive performance to C++ can only be reached if Rust libraries access the services in the same low-level manner. The developer should be able to choose whether memory allocation or the standard library is necessary. It has hence been important for me to avoid dependencies on *std* features from my crates. In cases where this would add convenience, additional implementations with standard library types can be explicitly switched on.

Based on the idea of a `l4re-core` directory, I decided to arrange my libraries in a directory called `l4rust`. The contained libraries are treated as individual packages and the programmer can choose which parts are required. Choosing a sensible distribution of functionality among L4 Rust crates was not obvious from the beginning. In my first draft I favoured a design where each functionality would reside in a particular crate, allowing the programmer to select which features are required (task and thread interface, memory allocation, etc.). This would allow for smaller binaries, but makes programming with the multitude of interfaces inconvenient. In my final design I chose to create an `l4-rust` and `l4re-rust` library, matching roughly the structure of the `l4sys` and `l4re` packages. For the `l4` crate, I followed the common practise in the Rust ecosystem, where the raw bindings to the C library are in a separate package that is used as a dependency. The library's name is suffixed with `sys`. The *l4-sys* crate therefore contains both reimplementations and bindings to existing C wrapper functions as well as reimplementations of functions accessing the basic interfaces (including task, factory, etc.). The C interface types and functions in the *l4-sys* crate can be generated using a helper program or library that calls Bindgen from a build script before the compilation of the actual library.

4.2.1 L4 Library Split

The `l4` crate makes use of the sys crate and introduces safe abstractions over some of the functionality. The most outstanding ones are the capability API, the UTCB types allowing serialisation and deserialisation of data into the virtual registers and the IPC framework, explained later. It also exports the raw definitions from the *l4-sys* library to save the programmer the double dependency declaration.

The *l4re* crate contains both the FFI definitions for types and the abstractions built on top of it. A large part of this library consists of client and server implementations and interfaces for L4Re services written in C++. Most of them are provided as header-only implementations; only a fraction is contained in compiled library code. Given that one of my goals was to introduce a C++-compatible IPC framework for Rust, I decided to port the interface definitions, instead of binding to the provided C wrapper functions.

4.2.2 Inlining vs. Reimplementing

Most of the functionality of the C libraries is written in header files, which is especially true for inline functions and templated functions and classes in C++. Because the code of an inline function is inserted at the position the function was called, it does not result in a public symbol in the binary. Therefore, languages other than C/C++ cannot use them. To safe time and to avoid the introduction of new bugs, I first attempted to wrap each function in a non-inline C function. These were outsourced in a separate library called `libl4re-wrapper`. The *l4-sys* crate then provided an inline Rust wrapper calling the C version from the new library.

Listing 4.2: Example For A Function Reimplemented In Rust

```
1  #[inline]
2  pub fn msgtag_has_error(t: l4_msgtag_t) -> bool {
3      (t.raw & L4_MSGTAG_ERROR as i64) != 0
4  }
```

While wrapping functionality avoids introducing new bugs, there is a certain overhead when calling half a dozen functions to set up and carry out the IPC. This especially applies for small functions that are called repeatedly. Most of these functions are essential for IPC operations, on the critical path and frequently called. Listing 4.2 shows how easy a reimplementation for them can be. The reimplementation also helps to avoid the usage of unsafe Rust to call into a C library. Most of the `l4sys` helper functions deal with bit-wise operations for which the Rust syntax is almost compatible with C. The porting work therefore often reduced to adjusting namespaces.

Rewriting functions becomes problematic when it comes to inline assembly or GCC-specific functionality. An example are functions such as `l4_ipc_call` or `l4_ipc_wait`. In the overall IPC setup process they appear infrequently. I therefore wrapped these in a separate C library as extern C functions. This helped to avoid the usage of inline assembly, which is not available on stable Rust. To avoid name conflicts and to mark functions as wrapped, I used the suffix `_w` for their redefinitions.

Some l4sys functions use GCC-specific features, such as functions to count the length of a string or even to allocate memory. This is necessary to avoid the usage of the C library which uses the l4sys package as a dependency. By restructuring code, I was able to avoid the usage of most of the GCC intrinsics. Some functionality such as C string comparison was reimplemented in Rust.

Parts of the l4sys C library provide an API for basic system services, such as the interfaces for accessing and managing factories, tasks and threads. These operations are simple IPC messages to the kernel.[6] The communication scheme is a recurring process of filling the message and buffer registers, making a call and working with the result. Repetitions can easily be mitigated by Rust's macro system which is why I decided to avoid the wrapping overhead for these interfaces and reimplement the functions in Rust. An advantage is that some of the functions in the *l4-sys* crate can be called from safe Rust.

4.3 Rust Abstractions

To provide a safe and idiomatic usage of the L4 services, I implemented a few basic abstractions to avoid the usage of unsafe FFI code. This section shall give a short overview of two concepts that are required to understand the IPC framework implementation.

4.3.1 Error Handling

The L4 crate builds on the error handling facilities of Rust, provided by the *core* crate[7] (compare Section 2.2.1 on page 13). IPC errors can have two origins: the thread control registers and the message tag. If an IPC operation was incomplete or interrupted, the error code is written to the thread control registers by the kernel. User processes can reply with an error code by using the label of the message tag. L4Re error codes are negative integers. The `l4::Error` type allows for conversion from and into an integer-based error code, applying the error bit mask if required.

[6] They can be interposed as well.

[7] Rust developers will know the error types from the standard library. These are re-exported from the underlying *core* crate.

Listing 4.3: L4Re IPC error handling using the C API

```
let error = l4_ipc_error(l4_ipc_call(...), l4_utcb());
if (error) {
  println!("Got error code {}", error);
  return;
}
```

Listing 4.4: Idiomatic error handling using the `result` of the `MsgTag` type

```
let _ = l4::ipc::call(...).result()?;
```

Listing 4.3 and 4.4 compare the C API error handling strategies of the C IPC bindings with the idiomatic version of Rust. The C version uses the `l4_ipc_error` function to extract the error from the message tag returned by the IPC call. The programmer needs to take care to interpret the numeric error codes manually.

The idiomatic version of Listing 4.4 uses Rust types exclusively. It starts with the `call` function that returns a `MsgTag` object. The message tag type provides a `result` method that extracts the error code of the message if present and converts it into a Rust enum. Apart from checking that the error code represents a valid enum variant, this step does not add additional runtime overhead. This is due to the internal representation of the enum as an integer.[8] A Rust developer will directly recognise the default error handling strategy of propagating an error upwards using the question mark operator or unwrapping the value if no error was present. The performance cost of this is as high as the manual check for an IPC error and is still left optional. If a programmer decides to omit error handling by the calling `unwrap()` method on the result, the program would abort, printing the error name (e. g. `InvalidMem`) along with the source file and line.[9] This is thanks to the error type implementing the `Display` trait, printing the enum variant name instead of the underlying IPC error code.

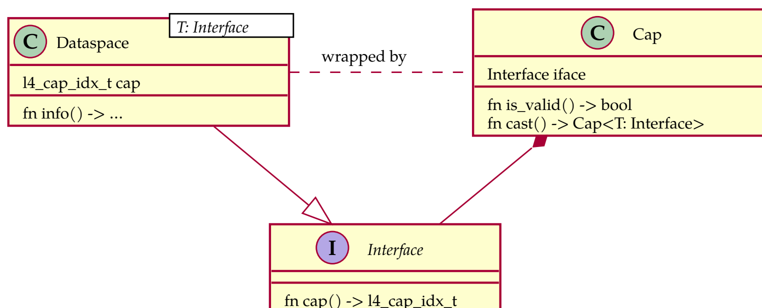
4.3.2 Capabilities

Capabilities play a central role in L4Re for interaction with the world outside the current task. Since capabilities are an index into a per-task table of kernel objects, the C type is only a type alias to an integer.

[8]This property is achieved by annotating the enum with `#[repr(i64)]` and by the usage of functionality from the *num-traits* crate.

[9]Normally it would also print a backtrace, but this is not available for L4Re yet.

Figure 4.1: Schematic overview of the capability type architecture



The object behind a capability can be accessed using the normal IPC mechanisms and is only distinguished through the protocol it supports. The protocol is hence the key property of a capability. The *l4* crate splits a capability into two parts: the interface defining the protocol and the `Cap` type implementing actions on the capability itself, as shown in Figure 4.1.[10] A protocol interface is a wrapper struct around the raw capability index, but may contain more fields. It provides the methods to carry out the protocol actions through IPC calls on the index.
The `Cap` type defines common operations such as checking the validity of a capability index or casting the internally used protocol implementation to a different one. For the `Cap` trait to be able to use the capability index of the protocol, the Interface trait has been introduced. It serves two purposes: it provides a common marker trait for IPC protocols and it allows access to the underlying capability index. The diagram illustrates that the `Cap` type is a wrapper around the `Interface`, as is the `Dataspace` implementation around the `l4_cap_idx_t` type. The `Cap<Dataspace>` type is a zero-cost abstraction because of `Cap` and `Dataspace` wrapping capability index and due to Rust using static dispatch.

Listing 4.5: Capability Type Demonstration

```
1 use l4::Cap;
2 let c: Cap<Untyped> = l4re::env::get_cap("test").unwrap();
3 assert_eq!(c.is_valid(), true);
4 let ds = c.cast::<l4re::Dataspace>();
5 assert!(ds.info().is_error());
```

Listing 4.5 shows the `Cap` type in action. It starts by querying an untyped capability from the initial set of capabilities that each L4Re task has. The `Untyped` interface is an empty protocol with no functionality. Line 3 and 4 demonstrate the call of methods

[10] The UML annotations **C** and **I** for interface and class should be substituted by trait and struct for the given example.

from `Cap` which use the capability index stored in the protocol interface. The cast on line 4 deserves special attention: because an `Interface` has defined behaviour, the interface can be casted from one type to another. The resulting `Dataspace` interface provides, among others, a method to query information about the dataspace behind that capability. Since no dataspace has been requested so far, this operation must fail, which is asserted.

The `Cap` abstraction has the advantage to be easily extensible. For instance, the *l4re* library contains an extended version called `OwnedCap`, which is able to use the L4Re capability allocator to automatically allocate a new capability index and drop it when the owned capability goes out of scope. Thanks to Rusts ownership and type system, this is as efficient as calling the `alloc()` and `free()` functions of the `cap_alloc` interface from C because these calls are inserted during compilation.

The design is similar to the C++ version which separates generic capability functionality from the protocol implementation [36]. Instead of wrapping types within each other, the C++ version uses inheritance to implement smart capability functionality, similar to the `OwnedCap`.

5 IPC Framework Implementation

L4 IPC primitives are designed to be minimal to maximise the call performance. As a result of the low-level steps, the interaction protocols can grow complex and thereby expose a new surface for introducing bugs in the communication architecture of the components. Different solutions exist to mitigate the complexity of protocols, mainly through the usage of IDL, for instance in Barrelfish [9]. IDLs have the downside that they are defined out-of-source. In today's L4Re, this has been integrated more tightly with C++ programming language by creating a framework capable of defining the communication as methods invoked on an object, similar to a remote procedure call (RPC). The first advantage of the framework is that it assigns names to the protocol actions as if invoking a local method. The second is that arguments and return values are automatically serialised and deserialised into the registers of the UTCB.

From a simplified view, the user defines a templated C++ class that declares the methods to be present for the communication. Then the framework will generate the client side and some parts of the server side code [19]. Server interfaces are defined in include files, making it easy to share them between C++ code. However, to expose the C++ API to other applications, a C interface wrapper needs to be written for each method of the IPC interface. This makes the usage from other programming languages cumbersome and also negates some of the performance benefits that can be applied when using the framework from C++ directly.

Instead of using the indirection via a C library, I decided to build a new framework in Rust from scratch. This is possible since communication takes place via the message and buffer registers, requiring a fixed "ABI" between the communication partners[1].

The following section starts with a short overview of the C++ framework and explains some of the details relevant for Rust interoperability. Next, channel-based and macro-based interface definitions are compared. This is followed by the explanation of the serialisation functionality that forms the foundation of the framework. Afterwards, different approaches for the implementation of a server loop and the final loop design are discussed. The last two sections discuss the automated export of Rust interfaces to C++ and the testing of the Rust IPC framework.

[1] It is comparable to an ABI for RPC.

5.1 A Brief Overview of the C++ Framework

The C++ framework defines each IPC call (and its receiving counterpart) as a distinct function in a protocol. It takes care of writing and reading from and to the virtual registers of the UTCB transparently. Listing 5.1 shows the example client/server calculator interface from the L4Re snapshot. An interface inherits from the `Kobject_t` type, whose parameters define properties of the interface, as shown on Line 1. Most important is the third template argument, the protocol ID used for the label in a message tag of an IPC call. An optional fourth parameter can be used to describe the receive slots for this protocol on the server side. This parameter can be omitted because no Flexpages are sent.

Listing 5.1: Calculator interface in C++ from the L4Re examples

```
1  struct Calc : L4::Kobject_t<Calc, L4::Kobject, 0x44> {
2    L4_INLINE_RPC(int, sub, (l4_uint32_t a, l4_uint32_t b,
3                             l4_uint32_t *res));
4    L4_INLINE_RPC(int, neg, (l4_uint32_t a, l4_uint32_t *res));
5    typedef L4::Typeid::Rpcs<sub_t, neg_t> Rpcs;
6  };
```

The interface methods are defined on lines 2–3 using macros and can be chosen depending on the requirements on the interface method. The `L4_INLINE_RPC` is the simplest version, defining only the name, input and output parameters and the return value. The first macro argument is the return value, the second the method name, followed by a list of call arguments enclosed in parenthesis. The C++ framework defines mutable pointers and references as output parameters and by-value or const pointers and references as input parameters.[2] This means that a callee of the `sub` function needs to provide two values for the subtraction and a pointer where the framework will write the result of the IPC operation to.

To distinguish between different operations, the framework uses opcodes, which are written as a first parameter to the message registers. In the example from Listing 5.1, these are auto-enumerated and hidden behind the `Rpcs` type definition of line 4, which is responsible for the enumeration. The RPC macros do not create method definitions from the passed arguments, but define inner structs that encode the argument types and order. These are then registered with the Rpcs type. By creating public members of these structs with the name of the corresponding method, it is possible to redirect calls on the them to code implementing the serialisation and call functionality. For the above example, there is a `sub` member of type `sub_t` that implements the call operator.

[2]There are also specialised types for parameters used for input and output.

The serialisation framework is optimised for maximum performance. In the C data structures, the finest granularity for message register access is a machine word. In contrast, the C++ framework packs data more tightly, with the alignment of the types dictating the next spare memory region. The aim is to reduce the memory footprint and the costs for accessing the memory. This means that for line 2 of Listing 5.1, the two arguments to `sub` fit into one word on a 64 bit system.

The macro and the template meta programming derive the complete client-side part of the communication protocol, so that the interface can be used like a local data type. On the server side, the service type inherits from the `Epiface_t` type and implements the method dispatch for incoming messages automatically. The implemented server-side names of the IPC method handlers are prefixed with `op_`. The developer only needs to implement the method almost like every other method: input parameters are used and results written to the supplied output parameters.

5.2 Rust Interface Definition

The previous chapters have introduced different methods for specifying IPC protocols, including IDL, channels and in-language code generation. As a consequence of the L4Re IPC evolution, I decided against the usage of an IDL. The introduction of a new DSL is problematic because it would need adaptation both for Rust and C++ to reach the aim of compatibility.

In this section, I will introduce the macro-based IPC framework and discuss its evolution from a channel-based approach.

One aim during the implementation of the framework was its independence of the standard library to avoid a dependency on Uclibc and thus on a variety of standard L4Re services. This includes, for instance, allocation or file system access. Given that the framework can be used to implement the backend services for this functionality, it makes more sense to not rely on *std* at all.

5.2.1 Channel-based Communication

Rust channels and L4Re IPC Gates share common designs: they transport data unidirectionally and require data to fulfill certain traits. Furthermore, the communication can happen synchronously in Rust channels,[3] matching L4Re's behaviour. Rust channels are by default agnostic about the data being sent, as long as it implements the

[3]The standard library ships both asynchronous and synchronous channels.

Send trait. To reach the aim of easy usability, it is necessary to restrict the channel to a protocol in order to enforce the correct types and order of arguments. This can be implemented using session types [14]. A session type leverages the build system to encode steps of a protocol in the type system. In Rust, this is achieved using generic arguments. Each execution branch that a protocol may take is represented by a type which holds generic references to the current step and to the subsequent ones, thereby spanning a type list. An important concept is the principle of duality, which demands that a protocol has a dual of itself that enables the receiver to understand the counterpart of the protocol. Applying this to L4Re, an IPC call can be represented as a type list containing the RPC arguments as list items and whose dual is the server side.

Listing 5.2: Excerpt of the type list implementation and demonstrative example

```
1  pub trait HasDual {
2    type Dual;
3  }
4
5  struct Sender<T, Next>(PhantomData<(T, Next)>);
6  impl<T: Serialisable, Next: HasDual> HasDual for Sender<T, Next> {
7      type Dual = Receiver<T, Next::Dual>;
8  }
9
10 struct Receiver<T, Next>(...);
11 impl<T: Serialisable, Next: HasDual> HasDual for Receiver<T, Next> {
12     type Dual = Sender<T, Next::Dual>;
13 }
14
15 struct End;
16 impl HasDual for End { type Dual = End; }
17
18 type CalcSub = Sender<u32, Sender<u32, End>>;
19 let receiver = CalcSub::dual;
20 //   ^^^^^^^^ -> Receiver<u32, Receiver<u32, End>>;
```

Listing 5.2 shows the application of session types to the sub method of the calculation server interface. The first building block is the trait HasDual which requires the implementer to specify its type dual on implementation. The impl blocks of Sender and Receiver specify their respective counterparts on line 6 and 11.

The Sender type is declared on line 5 and is a struct with two generic arguments, representing the current call argument type and a generic to the next list element. The Next generic is constrained by the HasDual trait bound, so that the principle of

duality is given inductively for the whole type list. The phantom members of the structs are used to inform the compiler that the generic arguments are required and are not dead code subject to be optimised away. `PhantomData` is a zero-sized type, making the enclosing struct zero-sized as well, so that the protocol specification does not take up any memory.

The `End` node declared on line 15 is special because it specifies itself as the dual. Since it lacks generic arguments, it encodes the end of the type list. A return value and operand code specification have been omitted for brevity reasons.

The duality property of each list node and its recursive structure allow defined type lists to be inversed to match their respective dual versions. The example type list for the subtraction operation of the type server is defined for the sender on line 18. The receiving dual is automatically derived on line 19 during compile-time.

The next step would be the extension of the channel API to let the channel enforce the protocol specification defined using the type list, as done in [14]. It turned out that this approach would be unergonomic to use due to the manual, repetitive interface usage. The user would still need to pass each call argument manually to the channel. Even though this uses safe Rust and the protocol would be enforced during compile-time, the overall usage pattern is close to that one of C++ streams, where all values have to be written to the UTCB explicitly by the developer [8]. Another issue is the location of the type lists for each IPC operation within the module hierarchy: the C++ framework uses inner types to hide away the implementation defaults. Rust does not offer inner structs, requiring a grouping module for all operations of a protocol. This makes the overall interface cumbersome to use.

5.2.2 Macro-based Interface Definition

Rust traits are used to define common behaviour across types. The most common usage is the definition of shared methods agnostic of the target type. A trait does not leak information about the actual implementation details, hence it is transparent to the callee of an object implementing a trait, where and how code is executed. This makes a trait a good fit to share protocol behaviour across clients and servers. The usage of such a trait can be made convenient by generating it through a macro. This idea was inspired by TARPC [32] which uses a macro with a custom language to define a service for RPC. The TARPC macro generates the corresponding trait that the user can work with.

For the L4Re IPC interface definition, I chose to follow the example of TARPC and implemented a 1.0-style macro as a façade to the more complex macro calls and type definitions. To ease the learning process, I kept the macro call syntax compatible

with Rust traits. An example is shown in Listing 5.3, matching the C++ version from Listing 5.1 on page 36.

Listing 5.3: Example calculator interface for a client/server calculation protocol

```
iface! {
    trait Calculator {
        const PROTOCOL_ID: i64 = 0x44;
        fn sub(&mut self, a: u32, b: u32) -> i32;
        fn neg(&mut self, a: u32) -> i32;
    }
}
```

The trait uses standard Rust syntax, but adds a few additional requirements on the definition. The first is the mandatory protocol ID specification on line 3. Furthermore, each interface method requires a mutable `self` reference, allowing state mutations in a server implementation. Empty interfaces are forbidden.

The declared methods are automatically enumerated and their assigned opcode is compatible to the C++ framework, as long as the method order matches the `Rpcs<>` order (see Section 5.1). Similar to the C++ framework, the opcode type can be overridden by an associated type called `OpCode` in the interface trait (not shown in the example).

Output Arguments and Error Handling Unlike C/C++, Rust does not use pointers or mutable references to write output arguments to. These are returned directly by the function. To propagate error state, it uses a `Result<T>` type,[4] to return either an error or a value. The methods in Listing 5.3 are declared without a result type, even though they are generated as such. The actual return signature of `sub` is hence not `i32`, but `Result<i32>`. This is added behind the scenes to allow for easier readability of the interface. I decided to omit this detail in the interface definition to keep it concise to read. In the retrospective, this introduces an inconsistency with the explicit notation of the mutable self reference which the implementer needs to specify repeatedly. In the future, it makes sense to require the explicit specification of the `Result` type in method signatures.
Both client and server side benefit from Rust's high-level error handling facilities and abstract from the integer codes used in L4Re. The conversion between the integer codes and the error enum is cheap because the error type of the *l4* library can be cast to an integer.

[4]The L4 crate defines a type alias: `type Result<T> = Result<T, l4::error::Error>`.

Client Generation The `iface` macro generates the full client implementation from the trait declaration passed to the macro. The client code is highly repetitive: write the opcode, serialise the call arguments, generate the message tag, do a call and read the return value. The methods can hence be generated and are stored as default implementations in the interface trait. Within each method, the first written value is the opcode which the macro derives from the method position. Next, the argument list is serialised and the IPC call is carried out. Afterwards, the replied data is deserialised and passed as return value back to the callee. Since the implementations are contained in the trait itself, implementing the client struct is an empty impl block.

Server Generation The server-side IPC has a similar repetitive structure, which is automated in a hidden and generated trait method called `dispatch()`. It reads the opcode from the first message register and uses it to dispatch to the corresponding method defined in the trait. Since the types of the arguments are known statically, the calls to deserialise them are inserted in the correct order for each method invocation in the dispatch method. A server struct needs to implement the public methods of the trait, as for any non-IPC struct. This also applies to return values (and errors, as explained above).

Deriving Clients And Servers Client and server implementations are usable as capability interfaces, as shown in Figure 4.1 on page 33. To accomplish this, they need to implement the `IfaceInit` trait, specifying how to set up a capability.[5] The consequence is that the developer needs to implement the `IfaceInit`, `Interface` and the IPC protocol trait. The repetitive character can be automated by a procedural macro attribute, as shown in Listing 5.4. The macro attributes transform the structs and insert the required members such as the capability index and implement all required interfaces. On the client side, the required input for this is the name of the protocol trait. On the server side, the macro attribute is able to derive all information on its own.

Listing 5.4: Automated Derivation

```
1 #[l4_client(Calculator)]
2 struct Calc;
3 #[l4_server(Calculator)]
4 struct CalcServer
```

Macro Implementation When I started the framework development, procedural macros were not stabilised yet. Therefore I based my first implementation on 1.0-

[5] `IfaceInit` extends the `Interface` trait.

style macros similar to TARPC [32]. The advantage was the quick definition and powerful matching of different syntax elements (e. g. type, literal, path). I split the implementation into smaller macros for the method argument serialisation, method body and trait generation and method opcode enumeration. This is brought together by the `iface!` façade macro.

As the code grew more complex, it turned out that the `macro_rules` have limited parsing flexibility and are not suitable for more complex input processing. The most apparent issue is that they are unable to deal with optional arguments properly. It is possible to match **zero or more** and **zero or one** occurrences of a syntax token, but to make use of an optional argument, each match arm of the macro needs to be duplicated for a version with and one without the argument present. The code duplication led to an interface macro with multiple hundred lines of code. This affected optional parts such as methods without return type, optional opcode type specification or even methods without arguments. Another limitation is that inputs cannot be compared against each other or against constants, preventing specialisation on different input arguments. For instance, when sending a Flexpage, the serialisation has to happen at the end of the used message registers, yet reordering is impossible here, since this particular type cannot be distinguished from other arguments. Another shortcoming is the problematic search for error causes within the nested macro structure. Any error in a deeper level is reported by the compiler as if occurring in the outermost level. This problem can only partly be resolved by viewing the expanded macro code because invalid macros generating syntax errors will not produce any output.

Procedural macros can solve this dilemma by providing the developer with the possibility to implement the macros directly in Rust. The stabilisation of this feature happened in the Rust 2018 edition, at the same time the IPC framework was implemented. Reimplementing the complete macro code would have been too time-consuming, so I decided to only reimplement the façade macro and pass an expanded version to the specialised macros. The macros of the deeper level always expect a complete specification of all features, with optional parts being filled by default choices. The `iface` macro façade is left with the task to check an interface for syntactic validity and whether it adheres to the additional requirements. A benefit is a more fine-grained error handling and the possibility to explain errors with custom error codes. It is also possible to report the exact error position within the macro input. An example of an error message from the `iface` macro is given below.

Listing 5.5: Example error message for an incorrect interface trait definition

```
error: First parameter must be &mut self
  --> calculator/interface.rs:14:16
14 |         fn sub(a: u32, b: u32) -> i32;
   |                ^
```

5.3 Data Serialisation

The serialisation functionality is the heart of the interoperability between Rust and C++ clients and services. In contrast to most serialisation frameworks, the data is transmitted using a private and lossless communication channel. The design maxim is therefore maximum performance; no attention needs to be paid on losing messages or transmission errors. The exact layout of the serialised messages is dictated by the C++ framework. The steps are as follows:

1. Cast base pointer of the registers to a `char *` and add the offset (of already used bytes) to it.
2. Align the pointer to the data type to be written.
3. Cast the pointer to a data-type-compatible pointer, i. e. `*mut l4_utcb_mr` to `*mut T`), dereference it and write the argument to it.
4. Add the length of the data type to the offset and repeat from the first step for the next argument.

This strategy makes sure that arguments are always written and read in an architecture-efficient and aligned fashion. The layout is similar to the C ABI of structs containing different data types. The templated function `msg_add` wraps these steps and can therefore serialise primitive data types. More complex types implement functions such as `to_msg` and `to_srv`, which decompose their data members into primitive types and call `msg_add` on each. The actual implementation is more complex, yet this knowledge suffices to implement a Rust counterpart. It consists of three parts: the `Serialisable` trait, the `Serialiser` trait and abstractions for reading/writing the message and buffer registers.

The `Serialisable` trait `Serialisable` is an empty marker trait marking a data type as serialisable. A data type is `Serialisable` when it can be written to a memory region without depending on additional state, so that it is valid within a different address space. Additionally, it needs to have a C++-compatible counterpart. The first condition is a stricter version of the `Send` trait, demanding movability across threads. As with `Send`, This marker trait is unsafe and the implementer must make sure that the conditions hold. It is automatically implemented for all primitive Rust data types, except for char.[6]

[6] The rust `char` type is a Unicode character. While its maximum length is known at compile time, its actual length is not. It is hence not supported and a string of length 1 can be used instead.

Virtual Register Abstraction The serialisation steps described earlier deal with pointers and pointer arithmetic and are hence using unsafe Rust. To minimise the potential for memory safety violations, I created a wrapper for write and read access of the virtual registers . It offers methods to read and write data types which implement the `Serialisable` trait and accounts offsets transparently. As in the C++ version, bound checks are inserted and converted into Rust error types. Due to the accounting, the register abstraction also knows the number of words and items in the registers and can be used for generating the message tag for an IPC call.

The `Serialiser` trait The `Serialiser` trait is implemented for all types sendable with a message. It contains a read and a write method whose arguments are the virtual registers to which they have access to. For primitive types, the trait is implemented automatically and contains simple delegation calls to the write and read methods of the virtual register abstraction. Complex types can be decomposed into types that implement `Serialisable` and are thus already known to the framework.

Listing 5.6: `Serialiser` Implementation for `Option<T>`

```
1  unsafe impl<T: Serialisable> Serialiser for Option<T> {
2      #[inline]
3      unsafe fn read(mr: &mut UtcbMr) -> Result<Self> {
4          let val = mr.read::<T>()?;
5          Ok(match mr.read::<bool>()? { // Option is valid?
6              true => Some(val),
7              false => Option::<T>::None
8          })
9      }
10
11     #[inline]
12     unsafe fn write(self, mr: &mut UtcbMr) -> Result<()> {
13         match self {
14             None => { // write "empty" Opt<T>
15                 mr.skip::<T>()?;
16                 mr.write::<bool>(false)
17             },
18             Some(val) => {
19                 mr.write::<T>(val)?;
20                 mr.write::<bool>(true)
21             }
22         }
23     }
24 }
```

Listing 5.6 shows the implementation of the Serialiser trait for the non-primitive Option<T> type. Rust allows to add additional trait implementations to a type outside the module or crate that it was defined in. This implementation will only be accessible from the same crate. This works in this scenario because the framework is implemented in one crate.

The compatible type from the C++ framework is Opt<T> and its struct layout dictates the serialisation and deserialisation code shown in the listing. It consists of a value and a boolean indicating its existence or absence. The read method therefore reads the value and the boolean, both implementing Serialisable, from the message registers and returns a Rust Option depending on the boolean flag. The write works in the opposite direction: if no value is found, the value serialisation is skipped and the boolean written, otherwise the value is written first, followed by the boolean value true.

Table 5.1: Overview of type mappings from Rust to C++

Rust Type	**C++ Type**
u8	unsigned char
i8	signed char
u16 – u64	l4_uint16_t – unsigned l4_uint64_t
i16 – i64	l4_int16_t – l4_int64_t
f32, f64	float, double
usize, isize	l4_umword_t, l4_mword_t
bool	bool
()	void
Option<T>	Opt<T>
Cap<T>	Cap<T>
String	String<>
&str	String<>
Vec<T>	Array<T>

Table 5.1 shows all Rust types currently supported by the serialisation framework and their C++ counterparts. Vec and String are heap-allocated data structures. Their usage is convenient and safe because the data is copied into their storage and hence owned by the callee. This comes with an increased overhead due to the allocation.

String and &str are both Rust string types. A Rust string is guaranteed to contain valid UTF-8 text, which is checked on its creation. Internally, it consists of the length and a reference to the memory where the data is stored. This is where the difference between both lies: &str is a string reference which can point to any region, including to the stack. A String is heap-allocated and an owned data struct. It is allocated on creation and freed as soon as its binding goes out of scope. Receiving a &str via an

IPC operation is unsafe because it references the data from the message registers and will be overwritten on the next IPC call. It is the users responsibility to copy the string, e. g. to an array on the stack or to an allocated `String`. It is advisable to evaluate the usage of an unsafe wrapper type for `&str` in future versions of the framework to reflect this. The `String` type is not affected because it always copies the data to its heap storage.

5.4 Server Loop

Microkernel services share a common structure across all protocols: they wait for incoming messages, call the server logic, reply and wait again. This can be described as an endless server loop, receiving and dispatching messages and replying with the result from the IPC server implementation. In L4Re this is a loop with recurring reply-and-wait operations, which is a joint system call. It uses an open wait operation to receive messages from any senders and uses the label attached to each message to identify the IPC gate on which it was received. After reading the message from the UTCB, the server logic is executed, the reply is written back and the loop will start again with the joint reply-and-wait.

The first step before launching a loop is the label registration with the kernel. As explained in Section 2.1 on page 8, a label has the size of a machine word, the same size as a pointer. This fact is used by the C++ framework to reinterpret the label as a pointer to a server object. This works because all servers are derived from the common `Epiface_t` type. The upcasting to `Epiface_t` is enough to call the virtual `dispatch()` function. The usage of arbitrary pointers without guarantees about the target object is unsafe in Rust and considered not idiomatic. During the development, I came up with two different versions of the server loop and I will discuss both in the subsequent sections. For both implementations, I decided to integrate the implementation for registering a new server object with the loop in the loop itself.

5.4.1 Vector-based Service Registration

Vectors store items of the same type (and size) on the heap and grow through reallocation. To allow different types to be contained in a vector, objects can be cast into trait objects, wrapped in a `Box<T>`. A `Box` is a fat pointer with meta information about the object stored in the pointer and a reference to the heap-allocated memory that it refers to.

The server loop in Listing 5.7 contains a vector of objects implementing the `Dispatch` trait. Since it acquires ownership through the usage of a `Box`, the server loop can make sure on registration that the server objects live as long as the loop. When a newly instantiated IPC service is registered, the vector of the loop takes ownership of the object, casts it to a `Dispatch` trait object and registers the vector index of the object with the kernel as a label for the corresponding IPC gate. The `Dispatch` trait contains the implementation to dispatch an incoming message to a specific IPC interface method of the trait.

Listing 5.7: Server Loop With Vector-based Registry

```
pub struct Loop<Hooks: LoopHook> {
  thread: CapIdx,
  utcb: *mut l4_utcb_t,
  servers: Vec<Box<dyn Dispatch>>,
  hooks: Option<Hooks>,
}
```

On message arrival, the kernel passes the vector index of the server object that was registered for the IPC gate to the server loop. The loop uses the index into its vector of server objects and retrieves the box pointer. On dereferencing the boxed trait object, the generated dispatch method is invoked and delegates the received information to the RPC handlers.

Aspects like the send/receive timeout, the actions on application or IPC errors can be customised by server loop hooks. The `Loop` struct of Listing 5.7 is parameterised over a `LoopHook` trait. They allow the customisation of the loop behaviour without reimplementing its logic. The `LoopHook` trait contains default implementations for general IPC and application error handling as well as buffer register setup. These are used if no custom hooks are specified. The default handler does not keep state and hence are implemented on a zero-sized struct. This allows the hook implementer to keep state if required, e. g. to count the number of failures of a registered server object to implement a fault tolerance strategy.

Using a vector keeps the implementation lean and in safe Rust, but has two major drawbacks:

1. Each object registration and deregistration needs to touch heap memory. When an object is inserted and the vector's backing memory is too small, a new vector is allocated and the elements are copied. This introduces an unpredictable overhead from the server loop's perspective. Object deletion is equally problematic, since vectors are contiguous and deleting an object in the middle would mean relocation, copying and changing all registered kernel labels.

2. By using `Box` and `Vec`, the application developer is forced to use the standard library, pulling in a lot of dependencies. It is helpful to recall that the `l4sys` (C) package is used to implement backend functionality for the `libc`, on which the standard library is built in turn. Avoiding a dependency on dynamic memory hence improves flexibility and makes the framework applicable to low-level services.

5.4.2 Pointer-based Service Registration

The drawbacks of the vector approach led to a new design early in the development phase. As explained before, C++ server objects share the common parent type `Epiface`. It is therefore possible to reinterpret the incoming label as an `Epiface` pointer and using the dispatch functionality without knowing the exact server implementation. Because the dispatch method is virtual, simply upcasting the pointer works. When a virtual method is invoked on the parent pointer, the vtable is consulted for the actual method pointer. Rust in contrary favours static dispatch and does not declare functions as virtual. Whenever dynamic dispatch is required, an object reference is cast to a trait object which is a fat pointer containing the pointer to the object and the pointer to the vtable. This means that methods are only virtual when explicitly requested and have no runtime overhead otherwise. Due to this optimisation for the general case of static dispatch, using virtual dispatch is slightly more expensive because a trait object is twice the size of a normal pointer: the pointer to the object in memory and a pointer to the vtable. Since the label used by the kernel for sender identification is only 64 bit in size, this poses the question how to point to the object and the vtable at the same time.

During the analysis of the problem, I realised that a trait object is not required for this task because there is exactly one target method in the `Dispatch` trait which contains the information about where to dispatch an incoming message to. It is therefore feasible to keep only the function pointer to the dispatch method, eliminating the need for a fat pointer. A problem with this approach is that the dispatch method still needs a reference to the server object, i. e. the self-reference, to be able to call other struct methods. Through a trick, we can obtain two pointers out of one label by placing the wanted function pointer as a first member of the data struct. Assuming no reordering of members takes place, the address of the object will match the address of its first member. By assuming this memory layout, the received label from the kernel can be cast both to a `c_void` pointer and a pointer to the first element of the struct, which is a function pointer. This approach requires three assumptions to be made:

1. The type information of a server implementation is recoverable from an untyped pointer, as which the label registered with the IPC gate can be reinterpreted.

2. The layout of the server object in memory must be predictable at compile-time and the function pointer needs to be the first member of it.

3. The object must not be moved to not invalidate the registered pointers.

To fulfil the first requirement, it is possible to leverage Rust's type system by creating a generic function which casts a pointer from an untyped pointer (`c_void`) to a generic one (`T`). The generic function is shown in Listing 5.8. As soon as a server implementation makes use of this function, it specifies `T` to be of its own type `Self`, causing the compiler to generate a specialised function. By taking a pointer to the function with specialised types, it is possible to recover the type information from an untyped pointer, as long as the function is associated with the server object that it belongs to (see requirement two).

Listing 5.8

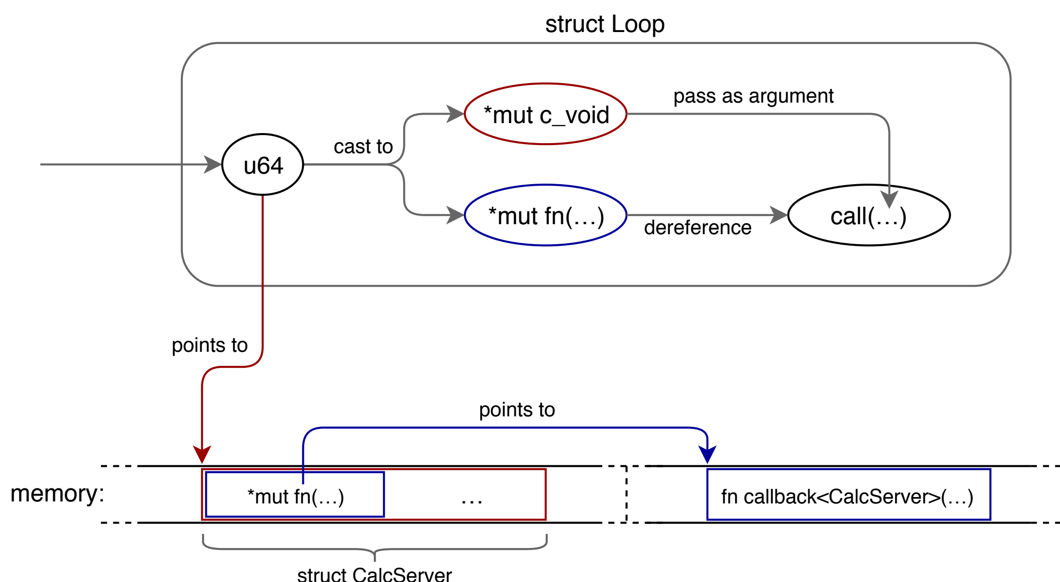
```
pub type Callback = fn(*mut libc::c_void, …) -> Result<MsgTag>;

pub fn callback<T>(ptr: *mut c_void, …) -> Result<MsgTag>
      where T: Callable + Dispatch {
  unsafe {
    let ptr = ptr as *mut T;
    (*ptr).dispatch(…)
  }
}
```

With the introduced way to recover the type information from a raw pointer using the type system, we can focus on the remaining two requirements. For requirement two, the order of a struct must be fixed. This is not guaranteed in the Rust ABI. Through the usage of the C ABI, the layout can be set to the same order as in the source code. This is achieved with the `#[repr(C)]` annotation to the server struct.

A sketch of the main memory layout is shown in Figure 5.1. It visualises the realisation of the `CalcServer` struct and the generated `callback<CalcServer>`. The `CalcServer` contains a pointer to the specialised callback function. This function is able to recover the type information of the server from the `c_void` pointer. To document the special requirements on function pointer setup and memory layout, the unsafe `Callable` trait is used. Unsafe traits require special attention from the programmer and enforce an unsafe `impl` block. The documentation comments explain the conditions to be met for a type to safely implement the trait.

Figure 5.1 shows the process of the invocation of a user-registered protocol handler. On message arrival, the server loop receives a label attached to a message which it

Figure 5.1: Overview of the server object invocation from the server loop and the required memory layout.



has registered before with an IPC gate to identify the sender of the message (compare Section 2.1 on page 8). Since it registered the memory address of the server object with the kernel, it can now safely cast it to an untyped `c_void` pointer because the label is still valid due to requirement three. The address of a struct matches the address of its first member. Because of requirement two, it is therefore possible to reinterpret the received `u64` label as a double function pointer[7] to the callback function of the `CalcServer` struct. To start the message dispatch, the function pointer is dereferenced and called, with the `c_void` pointer being its first argument. Listing 5.8 showed that the untyped pointer is cast so that it serves as a self reference.

To fulfill assumption three, the server loop must ensure statically that the server object is not moved (including not freed). Otherwise it is impossible to provide a safe abstraction over this process and memory safety rules are violated. In comparison, the C++ framework does not make any attempt to guarantee the validity of the server object pointer and it is the responsibility of the developer to make sure that registered server objects are either unregistered or outlive the server loop and are not moved. In Rust, this problem can be solved by *pinning* an object in memory, e. g. by including a member type in a struct which cannot be moved. This is achieved using a struct member of type `PhantomPinned`. As long as an object is pinned, it is impossible to move or free it, which gives the safe abstraction the guarantee about the object presence

[7] A pointer to the function pointer contained in the server struct.

it needs. The `Callable` trait requires a pinned memory object which can be realised adding a member of type `PhantomPinned` to the interface type.

The introduced approach is efficient, but requires the developer to know about a few low-level implementation and memory layout details in order to use the server loop efficiently. Apart from being a hurdle when learning the API usage, it also increases the chance to introduce new bugs. It is hence good to hide this implementation overhead through the usage of meta programming. Listing 5.4 on page 41 introduced the `#[l4_server]` procedural macro attribute that derives implementations for some of the required traits. It also automatically implements the `Callable` trait, that is, it adds the correct function pointer at the first position of the struct and a pinned member. This way, the unsafe details are hidden from the developer.

The complex casting of pointers is not as intuitive as the vector approach without further explanation, but compiles to efficient machine code. The dispatch method of the server loop results only in a few assembly instructions, as shown in Listing 5.9. The references called `mr` and `bufs` are passed to the generated dispatch method of the server to allow access to the message and buffer registers.

Listing 5.9: Disassembled code with debugging information as displayed by *objdump*

```
      let handler = ipc_label as *mut c_void;
      let callable = *(ipc_label as *mut Callback);
      callable(handler, tag, &mut mr, &mut bufs)
1000d5e: 48 8d bc 24 80 00 00 lea   0x80(%rsp),%rdi
1000d65: 00
1000d66: 48 89 ee             mov   %rbp,%rsi
1000d69: 4c 89 ea             mov   %r13,%rdx
1000d6c: 48 89 d9             mov   %rbx,%rcx
1000d6f: 4c 8d 44 24 18       lea   0x18(%rsp),%r8
1000d74: ff 55 00             callq *0x0(%rbp)
```

Receive Demand Before a task can receive a Flexpage, it has to set up the buffer registers so that the kernel knows where to map the incoming Flexpage to. If the receiver fails to set up the receive slots (or receive windows) in advance, the IPC operation will fail with an error code to both the sender and receiver. The server loop must make sure that it sets up the buffer registers to specify the receive slots before it blocks with a reply-and-wait operation. Since it has a global view on all registered servers, it needs to set up the registers for the highest required demand for all methods of all registered protocols. Simple services can be implemented with a demand of zero receive slots, that is, services which do not map any Flexpages. More complex services as, for instance, a dataspace server needs to work with capabilities and needs to set

up receive slots upfront. The interface specification encodes the information about the receive slots in the types passed to the interface methods. That means that the demand is known at compile-time. Yet the C++ framework requires the developer to specify the demand manually. In its first version, the Rust framework followed a similar approach. Each interface required a static demand specification as an associated constant of the trait. This manual step resulted from the inability of Rust `1.0` macros to match and compare types, hence making counting specific type tokens impossible. Because of the reimplementation of the façade as a procedural macro, this restriction was lifted and the demand is derived by the framework in the current version. To provide a common API from outside to query this information, the `Demand` trait exists. The `#[l4_server]` proc macro attribute generates the implementation for `Demand` by using the hidden information from the interface trait. This way, the information is transparently passed to the server loop which sets up the buffer registers before each reply-and-wait automatically.

5.5 Interface Export

The previous sections showed that Rust provides the language features to build a framework for inter-process communication and that most of the low-level details can be hidden by either the type system or through meta programming. But even though the interface definitions are concise and easy to read, it is a hurdle for a non-Rust programmer to translate this into a corresponding C++ interface. Replicating interfaces is also a common source of bugs, especially since conventions and types differ between the frameworks. It also introduces the risk of updating only one interface and neglecting the other. The ideal solution is hence an automated export of the interface definition into a C++ header file.

In a first draft, I started to write a small prototype interface parser in Python that parsed the Rust trait definition. I quickly realised that this was going to cause problems with each Rust syntax element that was not known to my custom Rust parser. With the advent of procedural macros and their stabilisation in Rust 2018, it became necessary to parse Rust source code from the procedural macro implementation (see Section 2.2.3 on page 15). The macro author can use the *syn* crate for this task, which implements a Rust source code parser. I have already explained the necessity to reimplement the frontend parsing code as a procedural macro in Section 5.2.2 on page 41. Therefore, the complete code for parsing and verifying an interface was already in place. The missing bit was the export logic.

The export routines are best called from a build script, which is the common place for Rust libraries and applications to generate and write files. An idiomatic solution

would provide a builder type[8] that first configures the exporter and then generates the C++ interface header. This proved to be impossible with the current Rust and the BID integration. In this scenario, the interface parser and validator is located in a common library shared among the *l4_derive* crate and the library containing the exporter. Splitting this into three libraries is required because the Rust compiler does not permit the usage of public types and functions from a procedural macro crate; only macros can be accessed. In its current form, it is impossible to specify transitive Rust dependencies in BID which is why this approach was not pursued further. As an alternative, it is possible to rework the adaptation with custom registries which complement the central `Cargo` registry with a custom one. This feature was stabilised after the implementation for this framework was finished.

As a temporary workaround, I exposed the exporter as a procedural macro from the *l4_derive* crate. It parses the options passed to it and the interface from the given file name, generates the C++ header file and writes it to the specified output path. Adapting it to a builder pattern later on has been kept in mind during the implementation which is why the `iface_export` macro is small.

A few differences between the frameworks deserve special attention because of their influence on the resulting C++ header file. First of all, the specification for input and output parameters differs. While the Rust version specifies output parameters as direct return types of the methods, the C++ version expects output parameters to be passed as a mutable pointer or reference. Only references passed as const pointers are treated as input parameters. The Rust interface expects all parameters to be passed by value so that it can take ownership of it. This makes the programmer aware that a copy of the value is required if it is still used after the send operation. Output parameters are returned by value as well and are wrapped within a `Result` so that either a value or an error is returned. Since the C++ framework returns IPC arguments using the provided output parameters, the return value of the RPC function is unused and is thus used to return the error code.

The type names also need to be adapted in the conversion process. The challenge on the C++ side is to reference the correct type name from the correct namespace, requiring a mapping of Rust type names to C++ type names as well as the C++ namespaces. To allow the generated C++ header file to be used without modifications, a list of mappings of C++ types to their include files is kept as well. It is possible to extend these default mappings to also handle custom types.

Thanks to the *syn*[9] library, the exporter can traverse generic arguments of types recursively, allowing a complete translation of each Rust type.

[8]This refers to the builder pattern which is commonly used in Rust for object initialisation.

[9]See the introductory paragraph on procedural macros in Section 2.2.3 on page 15.

Listing 5.10: Build script example for the CalcServer interface

```
#![feature(proc_macro_hygiene)]
extern crate l4_derive;
fn main() {
  l4_derive::iface_export!(
    input_file:  "../interface.rs",
    output_file: "../cpp.h",
    // optional: name: "CppReplacementName", protocol: 0xca1c
  );
}
```

Listing 5.10 is a complete export build script for the Rust calculator interface which already served as an example in previous sections. In line 1, the unstable language feature for macro hygiene is activated. This is necessary because of the macro usage in a statement context. This contradicts my goal of using stable Rust for the framework implementation. Given that the export macro is a temporary solution and that interface exports are optional, I decided to make use of this unstable feature for it. A future solution using a builder pattern would not require macro hygiene to be enabled.

The comment in line 7 shows that this macro supports additional parameters, which optionally influence the C++ header file generation: a custom name (potentially better fitting the C++ naming conventions) and a custom protocol specifier. A custom protocol number is required whenever a protocol definition references a constant outside the interface. This cannot be resolved by the interface parser and can hence be overridden at this point.

5.6 Implementation Tests

This chapter showed that the implementation of an IPC framework has not been straightforward and that rethinking the overall design has been required. Refactoring phases carry the risk that new bugs are introduced into an otherwise working code base. To mitigate this, software is often tested with unit tests or integration tests. However, there is no public testing framework for L4Re available. In the beginning, I filled this gap by writing example applications to test and demonstrate certain aspects of the framework. The obvious issue is that examples are not suited for testing all edge cases of inputs and outputs for functions or components.

Rust comes with a default testing framework, integrated both into the compiler and into Cargo. Its tight integration makes it the de-facto standard for Rust. It can test functions within the crate (white box testing), outside the crate (black box testing)

and even tests example code from the documentation comments. It uses conditional compilation to compile the test cases with the test framework into an application binary which is then executed. Assertions, mainly through the `assert!` and `assert_eq!` macros check conditions to hold. If a condition is expected to be false, the function is annotated with `#[should_panic]`. The user can invoke the test suite by executing `cargo test` which orchestrates the compilation and execution steps.

Using this framework without modifications is impossible for two reasons. The first is that the assertions panic on failure. To allow the test framework to continue executing, each test function call is wrapped in a panic catching environment which aborts the backtrace generation and continues executing. Because the backtrace library has not been ported to L4Re yet, panics cannot be cought. The second reason is that `Cargo` executes the tests on the host by default. It offers the possibility to launch the binaries using a wrapper script, executing the tests within QEMU. Starting an application in a virtual machine with L4Re requires an entry in the `modules.list` boot configuration to specify the components of L4Re to launch for this application. When I started writing a script for the automation of this process, I discovered that L4Re lacks the ability of shutting down the system from user space. There is the platform API which specifies the shutdown action, but the functionality itself is not implemented.

I therefore decided to write a custom test framework. Its design goal was to retain compatibility with the existing Rust framework and that the implementation overhead is small.

Listing 5.11: Example test framework usage

```
1  tests! {
2    fn reimplemented__l4_is_invalid_cap__works() {
3      unsafe {
4        assert_eq!(
5            (l4_is_invalid_cap_w(L4_INVALID_CAP) > 0),
6            l4_is_invalid_cap(L4_INVALID_CAP));
7      }
8    }
9  }
```

Listing 5.11 shows the comparison of the `l4_is_invalid_cap` function with its C counterpart[10]. All tests are wrapped within the `tests!` macro. It gathers the names of all test cases and adds them to a list. Due to the simple design, there can be only one macro invocation per module in which all test functions need to be contained. The list of test functions is used by the main executor to run all tests in all modules.

[10]See Section 4.2.2 on page 30 for the discussion of reasons and the naming scheme.

Rust allows shadowing macro definitions which is used by the test application to redefine the assertion macros. Their substitutes behave like the original (compare line 5 of Listing 5.11). Instead of panicking on assertion failure, they return a `Result`. The main loop can then collect the results and print a test summary. This also preserves the line number and the error message of the failing test.

In the original test framework, tests return no value. This conflicts with with the redefinition of the assertion macros which abort the test by returning an error. The function signatures of the tests can be kept compatible to the original test framework by the usage of macro transformations. The `tests!` macro changes the function signature of line 2 of Listing 5.11 to return a `Result<(), String>` and extends the function body to return `Ok(())` after the successful run of a test. If in the future, the Rust test framework is ported to L4Re, changing the tests is a matter of removing the `tests!` macro and the shadowing assertion macros.

As already pointed out, it is currently impossible to shut down an L4Re system through the platform API. There are solutions to this using I/O ports on x86 from within QEMU [27], signaling QEMU to shut down the system. The solution introduced here is a temporary one and I therefore decided against implementing shutdown support for my test application. A similar effect can be obtained by executing QEMU through a wrapper script. The wrapper script queries the QEMU console output for a specific string that indicates the end of the test run and kills QEMU if this string was found.

6 Evaluation

The previous chapters have shown the applicability of Rust in low-level systems programming on L4Re. This chapter focusses on evaluating the framework against the design goals set out in the introduction. An exception is binary compatibility, which has been demonstrated in the previous chapters through the `CalcServer` example and the interface exporter. The following two sections will focus on the evaluation of efficiency and easy usability of the framework.

6.1 Execution Performance

An IPC framework is only accepted in low-level systems programming if it keeps the introduced overhead minimal. The baseline for benchmarks is the IPC framework of L4Re written in C++ that has been optimised for many years and is used in the majority of L4Re services.

The benchmarks have been conducted on an Intel® Core™i5-4590 quad-core processor of the Haswell line with a base clock frequency of 3.30 GHz running a 64 bit version of L4Re. This CPU features a maximum single-core-clock speed of 3.70 GHz. The benchmarks have been executed on an L4Re build of the 18.11 snapshot with a Rust compiler from Git[1] The measurements were run on a single core exclusively, thus measuring single-core IPC.

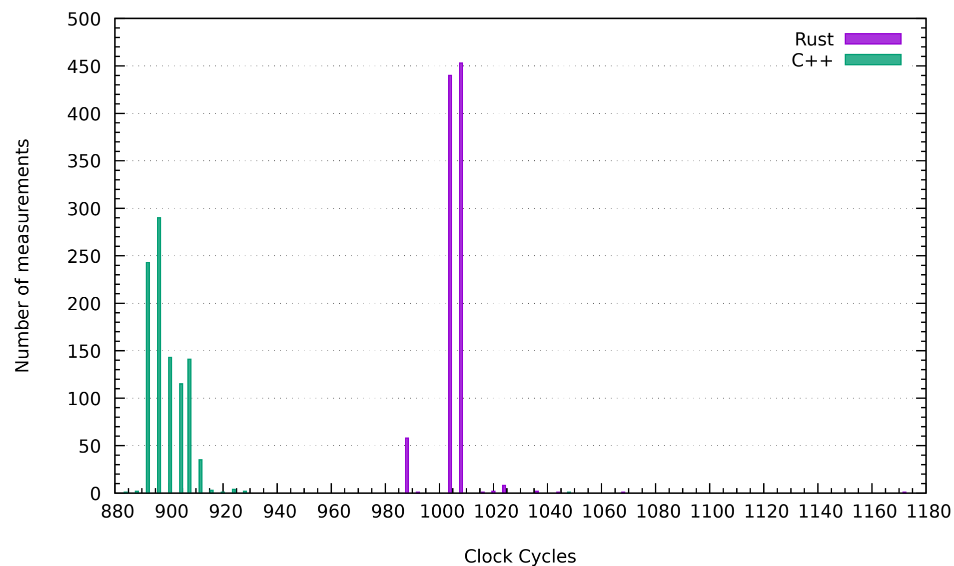
The common metric for performance in the L4 community is the clock cycle which is used here as well. It is measured using the Intel x86 `rdtsc` and `rdtscp` instructions which emit the clock cycles passed since the CPU was started. The `rdtsc` instruction does not enforce an order of the execution in the CPU pipeline. Thus there is no guarantee that the cycle counter is read before or after the event to measure [28]. Intel instead suggests to use the `rdtscp` instruction, which pseudo-serialises the instruction stream by waiting for previous instructions to be finished. Subsequent operations can already start executing and hence the read can happen later than the event to measure [28]. Taking precise measurements down to individual clock cycles is thus not realistic.

[1]The commit hash is `b8fa4cb31dcb2c3ed2c61f80ca6d0`, with the added patches for L4Re linking.

The l4util C library from the l4re-core directory provides an rdtsc function, but lacks one for rdtscp. I added this function using inline assembly. Rust offers support for both instructions in its standard library. Pseudo-serialising the instructions and reading the clock counter takes time and hence influences the measurement. This is why IPC round trip measurements were conducted with the microbenchmarking facilities disabled.

The relevance of the IPC framework performance can be better judged when determining the IPC operation costs beforehand. Figure 6.1 shows the distribution of execution time of a *send* operation from a client to a server. This test was executed 1000 times both for a Rust and for a C++ client/server pair.

Figure 6.1: Measurements of IPC cost for a *send* operation from a client to a server



The performance decrease from the C++ to the Rust version is roughly 13 % for the median speed. This can be attributed to the inlining of the l4_ipc_call and l4_utcb functions in the C++ variant, discussed in Section 4.2.2 on page 30. Rust has a few rare outliers up to 1172 cycles. The source of these remains unclear. The standard deviation for Rust is 7.87 cycles and for C++ 8.01 cycles and thus close to the average case. The accumulation of measurements around the average values is potentially caused by cache usage patterns. The rdtscp instruction seems to favour even numbers, which can be observed in the following benchmarks as well.

The subsequent benchmarks evaluate performance in a fine-grained manner. The measurement instructions were inserted into the C++ and Rust frameworks, so that

the individual steps can be compared across the benchmarks and were disabled for round trip measurements. The microbenchmarks are split into 12 steps. The first one is the call setup which initialises pointers to the message registers and other variables. It is followed by the serialisation of the opcode and the call arguments. After the message tag has been prepared in the IPC call setup, the IPC call is executed, triggering a context switch to the server. The time required to retrieve the server implementation from the IPC gate label to the dispatch method of the interface is called "loop dispatch". Afterwards the opcode and the method arguments are read. These are used to execute the interface method and then its return value is serialised again. The reply setup, the IPC reply operation and the return value deserialisation work in an analogous manner. All diagrams use the median of the clock cycle count.

6.1.1 Primitive Types

Primitive types are native Rust types with a fixed size and have a counterpart in C++. Examples include `u32`, `bool` and `f64`. As an exception, Flexpages can be treated as primitive too because they are serialised to two machine words on the sender and one word on the receiver side.

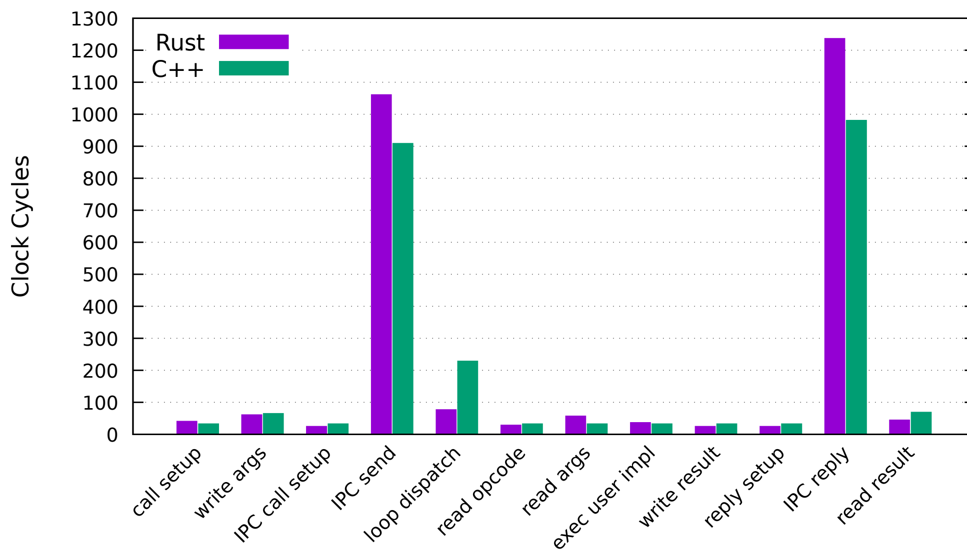
Figure 6.2 shows the measurements for a simple subtraction test. It is borrowed from the `CalcServer` interface that served as an example before. In this test, the client sends two integers to the server and expects the subtracted value as the result.

The dominating costs are the L4Re IPC operations, all other operations are below 200 cycles. The two implementations vary in individual steps, but the overall performance is comparable. The "loop dispatch" step refers to the dispatch from the generic server loop to the registered server object, see Section 5.4.2. The Rust version is fast, likely due to the optimisation described in Section 5.4.2 on page 48. Without further analysis, it is hard to find the reason for the slower C++ variant, but a possible reason could be the virtual method dispatch.

In general, the figure shows that the frameworks yield comparable performance with a slight variation among the different steps. The overall performance gap from Rust to C++ can be attributed to the additional IPC costs, compare Figure 6.1.

In the Flexpage benchmark, the client allocates a capability and requests a mapping of it to the server. The server receives it and checks the validity of the received capability. On both sides, the interface method signatures reference a specialised capability type such as `Cap<Dataspace>`. The conversion from and to a Flexpage is done transparently. On the server side, recovering the type from the mapped capability is a zero-cost abstraction because the type is known at compile time. Figure 6.3 suggests that the

Figure 6.2: Benchmark of IPC performance using primitive integer arguments



read operation of the Flexpage containing the capability process is not as efficient as it can be. This is left for future work.

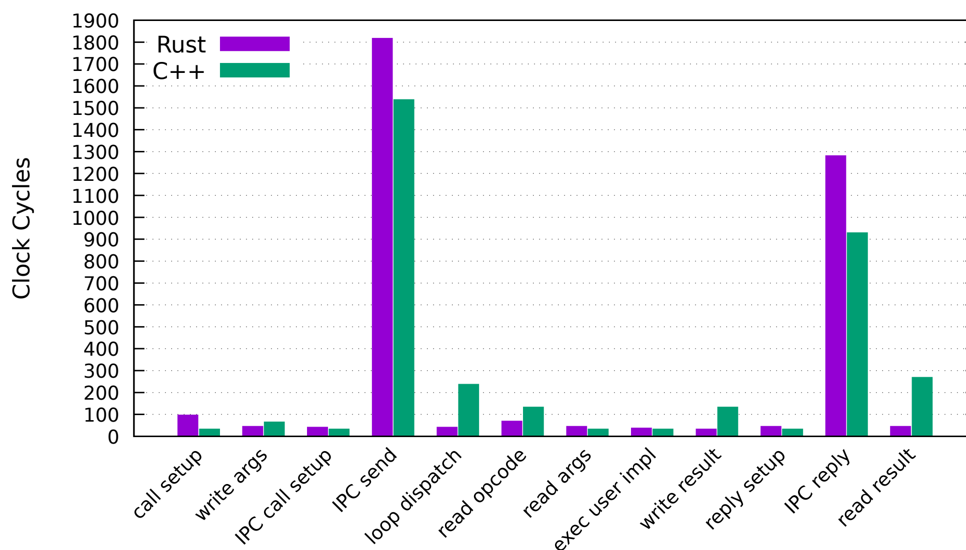
On the sender side, a Flexpage consists of a description of the object or memory page to map and the flags describing the object, taking up two words. On the receiving side, the server loop sets up a buffer register for each Flexpage to expect. Because of this simple memory layout in the registers, this benchmark was treated as a primitive type benchmark.

The benchmark of Figure 6.3 shows a similar pattern as Figure 6.2. Reading and writing the arguments is as fast as in the subtraction test which suggests that the type recovery of the `Cap<Dataspace>` object is without any runtime cost.

6.1.2 Strings

The C++ framework provides array types to transmit data in contiguous arrays of variable size using the UTCB. The array has to fit into the message registers, in addition to the other data that is transmitted with it. The serialised array starts with the length and is followed by the aligned data.

Figure 6.3: Benchmark of the performance of object Flexpage mappings

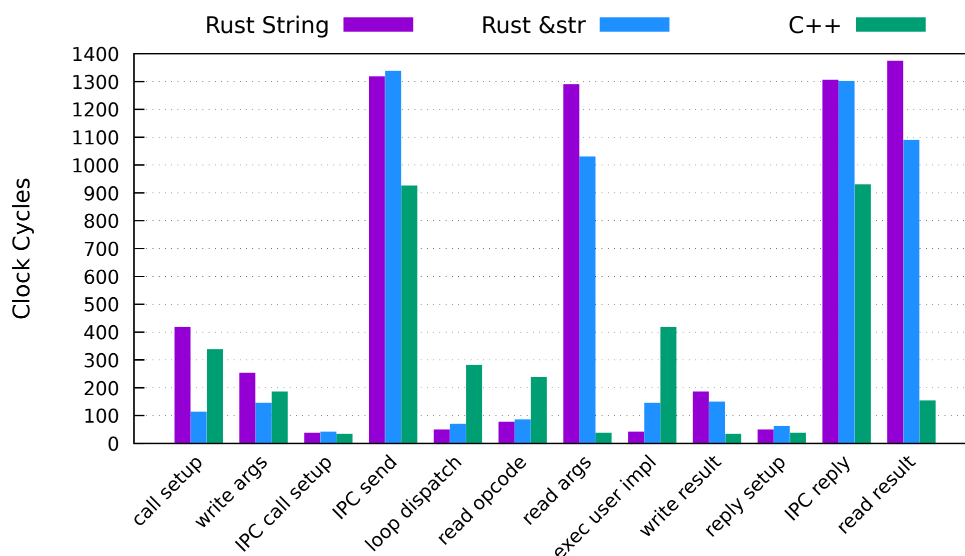


In both IPC frameworks, strings are implemented as a special case of arrays containing data of type `char`. In contrast to C strings, the trailing null byte is not mandatory, but suggested. The C++ IPC string does not own the data, but contains a pointer to it. On the client side, the framework takes care to copy the data into the message registers. On the server side, the string data needs to be copied from and to the registers manually. This is similar to the handling of `&str` in the Rust framework, see 5.3 on page 45. The usage of a `String` in Rust is more convenient because of its automated allocation management. `&str` only borrows existing data without allocation and thus requires more care by the programmer.

Figure 6.4 compares `String`, `&str` and the C++ string implementation against each other. The benchmark uses a string of 113 characters in length and sends it in a loop from client to server and back.

There is a noticeable delay for all tests in the "call setup" step, the time between the first measurement just before the call to the generated client method and the point where the argument serialisation starts. The setup code itself is short and hence the time is probably spent in the serialisation code afterwards. It is unclear why writing a `&str` is faster than writing a `String` because the code for it is identical. Deserialising the string from the UTCB is slowest for the heap-allocated String, but reading the `&str` is also slow. The C++ variant is order of magnitudes faster because it delegates the string copy to the user so that the user implementation is much slower. In contrast,

Figure 6.4: IPC ping pong benchmark of the Rust `String` and `&str` as well as the C++ IPC string type



the execution of the user implementation for the Rust `String` is fast because the copy operations are handled by the framework.

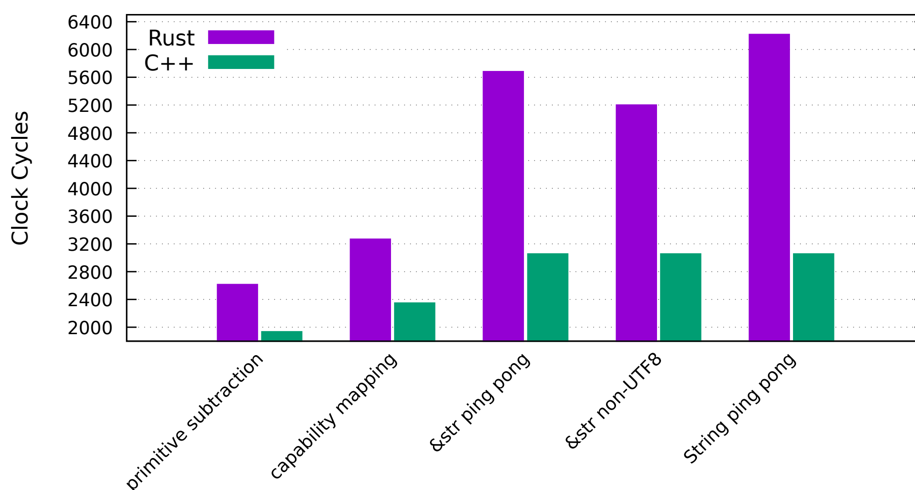
The numbers from Figure 6.4 show a clear performance penalty for using Rust strings. A possible cause could be the check for UTF-8 validity, but this can only be clarified by further experiments. A dedicated string type with relaxed safety and correctness guarantees can be another solution.

6.1.3 Round Trip Measurements

The round trip times of IPC calls offer a coarse-grained view on different operations of the framework. To take measurements, the library code can be left untouched, instead the CPU counter is queried before and after the call. This avoids the introduction of delays introduced by microbenchmarks.

Figure 6.5 compares the test cases from the previous sections. In contrast to C++, the Rust framework supports different string types, therefore the measurements of the C++ IPC string were replicated for easier comparison.

Figure 6.5: Comparison of the round trip times from a full IPC call for each benchmark



From left to right, we have a performance decrease from Rust to C++ of around 25,8 %, 39,2 %, 77,6 % and 103,4 %. The efficiency decrease for the primitive test cases is acceptable, given that the tests are compared to a C++ framework which has been optimised for years.

The string handling is inefficient, nearly doubling the time spend for the IPC. For investigating the performance loss, the mandatory UTF-8 check for a string during its initialisation was thought to be the most obvious cause. For comparison, `&str` has been benchmarked with and without this check. The performance gain for a string with 113 characters is only around 400 cycles so that it can be concluded that the majority of time is spend somewhere else. A possible reason could be an inefficient handling of the fat pointer by the framework. This investigation is left for future work.

6.2 Usability

To increase the acceptance of a new programming language in the L4Re ecosystem, the transition to it must be as smooth as possible and may not introduce incompatible work flows or result in incompatible software. It is also important that its advantages justify the introduction of new libraries or services into the system. The new compon-

ents should not result in an unmaintainable system because of high fragmentation among different programming languages.

This section covers the ease of use from multiple perspectives. It starts with the additional safety guarantees, followed by a summary of the idiomatic interface definition and usage. Afterwards, the exclusive usage of stable Rust is discussed, concluded by an evaluation of the L4Re integration.

6.2.1 Safety

Rust's borrow checker allows to reason about a number of safety guarantees during compile time. This includes, but is not limited to, memory safety and the prevention of data races in concurrent programs. The aim of each Rust library is the minimisation of unsafe blocks and functions. In low-level programming, this is impossible because accessing and mutating machine state is inherently unsafe [35].

The `l4rust` libraries, which also contains the IPC framework, are made up of about 3330 lines of source code, of which 262 lines (7.86 %) are unsafe or marked as such. The framework counts over 2460 lines, with 167 lines (6.79 %) marked as unsafe. These numbers are reported both by `cloc` and `cargo-count`. In certain situations, unsafe code cannot be avoided, as for instance in the server dispatch scenario (compare Section 5.4.2). In other situations, an unsafe trait or function is used to make the developer aware that certain conditions must hold for this type and violating them leads to undefined behaviour. There is no utility that allows for automated analysis of the different scenarios for unsafe code usage, but it can be assumed that the number of lines performing actual unsafe operations is even lower. Defining an IPC interface, deriving a client and implementing a service can be achieved using safe Rust exclusively (compare Listings 5.3 and 5.4). The developer can thus rely on the framework to never hand out invalid references, use uninitialised or freed memory or create memory leaks.

To avoid buffer overflows and accessing uninitialised memory, Rust inserts bounds checks. The IPC framework follows this pattern by adding bounds checks for UTCB accesses which take place during serialisation.

6.2.2 Interface Usage

Intuitive usability was a key goal from the beginning. It was therefore a logical consequence that an IPC interface is represented as a Rust trait, which is Rust's way of

defining common API. Both the client and the server share the trait definition, resulting in decreased maintenance cost for interface adjustments. Input and output parameters are passed like for local methods. Due to the trait bound on `Serialiser`, the compiler provides a meaningful error message to the user if an invalid type is passed. In contrast to the C++ framework, types do not differ between sender and receiver.[2]

IPC errors and framework errors are handled using Rust's standard error handling facilities. This means that errors can be propagated upwards using the question mark operator and handled where appropriate. Through the usage of Rust enums, errors can be represented as a human-readable mnemonic that eases debugging. Converting from an error code to an error enum is cheap and converting from an error enum value to an integer is a no-op because of its internal representation.

Extending the IPC framework is possible through the usage of the `Serialiser` trait. This can be done in safe Rust[3]. The server loop has been designed to allow to hook additional event handlers into the reply-and-wait cycles. Events include IPC or application errors. This facilitates the customisation of the server loop without reimplementing it.

The interface exporter accepts the specification of additional mappings from Rust to C++ types, as well as the specification of additional required namespaces and include paths. This allows for the usage of custom types in the interface definition and the automated export to a C++ header file.

The framework makes use of Rust macros. Repetitive work is avoided by the use of 1.0 macros and low-level implementation details are hidden by procedural macros (compare Section 5.2.2 on page 39). Especially the latter allow the programmer to focus on defining the IPC interface and implementing the server logic without the need to know about buffer register setup, type deserialisation and message dispatch.

A disadvantage of basing the framework on interface traits is that it requires an additional client and server struct which implement the trait. In C++, the interface serves as the client-side IPC object, avoiding the need to search for another name.

To allow shared server state, the interface demands a mutable self-reference as its first argument. This is inflexible because this forces the client object to be marked as mutable on initialisation, even though it does not contain mutable state.

[2] An example is `String<>` for an input and `String<char>` for its corresponding output parameter.

[3] The `Serialiser` trait requires the `read` and `write` functions to be marked as unsafe, but this serves a documentary purpose.

6.2.3 Stable Rust

The Rust team has chosen to release a new version every six weeks, supplemented with bug fix releases. Each new version adds a few new features or stabilises API from the nightly version. Breaking changes are only introduced in Rust editions. Basing the framework on Rust has been an ambitious aim. A few of the required features were unstable when the implementation started and were stabilised late or after the implementation work was done. In comparison to Clipsham [4] and Foellmi [9], Rust has reached a level of maturity with the 2018 edition. This also applies for the tooling of Rust. For instance, it is easy to add a custom cross-compilation target to `rustc` and to compile a new version of the compiler without prior knowledge.

The final design relied on the two features: *align offset* and *associated type defaults*. The first has been stabilised just after the benchmarks were taken. The second is easy to avoid with the minor drawback that the operand code for a protocol cannot be queried from the client or server struct.

6.2.4 L4Re Integration

The BID integration compromise has sufficed for the framework development and enabled substantial improvements in comparison to my previous work [12; 13]. Build scripts, conditional compilation and dependency management for crates from crates.io works and native Rust libraries are handled by BID.

A problem is the manual recompilation of transitive dependencies. For instance, an update of the `l4-rust` library requires `l4re-rust` to be recompiled. BID cannot handle this appropriately because it lacks the dependency information. Compiling Rust is slower than compiling a C++ program[4] and the manual recompilation of the L4Re libraries increases the waiting time.

The integration of the Rust programs into the L4Re ecosystem is smooth. Rust programs can interface seamlessly with C++ services and use C libraries. Exporting Rust definitions to C++ is easy and can be automated.

[4] This is hard to quantify, but has been a recurring topic on the official Rust blog and in the official Rust user surveys. See `https://blog.rust-lang.org/2018/11/27/Rust-survey-2018.html`.

7 Conclusion

In this thesis, a new framework for L4Re IPC communication, written in Rust, was presented and evaluated. The implemented framework builds on top of Rust's memory safety guarantees and extends them to safer IPC mechanisms. Its design goals include easy usability, binary compatibility to existing solutions and high efficiency. Rust's powerful macro system was used to provide a user-friendly interface with a flat learning curve. Binary compatibility has been ensured by implementing a custom test framework and by various example programs. The performance was evaluated through a number of benchmarks with different usage scenarios.

Rust has proven to be a language suitable for systems programming with performance comparable to C and C++ [5; 24]. Its advanced type system allows complex interactions to be expressed and to abstract from low-level details. The language contains high-level constructs such as iterators, closures, operator overloading and more that impose no or only a small runtime overhead [16; 23; 24]. This has been proven to be of great help during the implementation of the data serialisation into the UTCB registers. Together with the safe abstractions from the framework, example services were written without any of the typical memory management bugs.

The achieved benchmark performance of the Rust framework lies within the same magnitude of measurements compared with the existing C++ framework. It showed that the time for this thesis did not suffice to reach equal efficiency. This is not surprising because the C++ framework has been optimised for years by multiple developers. The performance decrease lies between 25 % up to 103 %, with string passing being the most inefficient operation (compare Figure 6.5). In future work, string serialisation should be revised and a type with less requirements on the string data should be supported.

The usage of C wrapper functions for preparing and sending IPC messages should be rewritten in Rust to allow for inlining of them. This could improve cache locality and would result in more efficient IPC calls. The usage of procedural macros offer the possibility to optimise the code generation further. For interface methods that take arguments with a size known at compile time, the bounds checks can be omitted, avoiding jumps in the instruction stream. In this scenario, the message tag can be precomputed as well. This would for instance apply to all methods of the commonly used dataspace interface, except the `copy_in` [18].

The integration of `Cargo` into BID is incomplete. `Cargo` is still not suited to be integrated into other build systems. One example is the missing possibility to add custom linker arguments containing spaces to the linker call. It is also obstructive for debugging that the library `libbacktrace`, which Rust uses to print a stack trace in the case of an error, cannot be compiled for L4Re. This is because Rust uses a different version than the L4Re snapshot. In future work, it should be tried to either publish L4Re's version or to extend that one of Rust.

The thesis showed that the stable version of Rust is applicable for the implementation of a low-level IPC framework for L4Re. Its high-level programming concepts and its ability to abstract from unsafe operations in a safe fashion allows the convenient realisation of microkernel services. There are still enhancement possibilities that suggest to continue the implementation of the framework. Further research should focus on leveraging Rust's type system to aid the developer with low-level memory interactions and to improve the static analysis of the interface definitions.

A References

[1] *An RPC framework for Rust based on tokio*.
Web: https://github.com/iorust/tokio-rpc (visited on 04/28/2018).

[2] Gui Andrade: *Storing Unboxed Trait Objects in Rust*.
Web: https://guiand.xyz/blog-posts/unboxed-trait-objects.html (visited on 12/14/2018).

[3] *Cap'n Proto for Rust*.
Web: https://github.com/capnproto/capnproto-rust (visited on 05/01/2018).

[4] Robert Clipsham: "Safe, Correct, and Fast Low-Level Networking". MA thesis. Glascow: University o Glascow, Aug. 2015.

[5] *Computer Language Benchmarks Game*.
Web: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust-gpp.html (visited on 02/20/2019).

[6] *Dr. Xu Zhongxing's speech: Fuchsia OS Introduction*.
Web: https://bzdww.com/article/163937/ (visited on 04/02/2019).

[7] Kevin Elphinstone and Gernot Heiser: "From L3 to seL4 – What Have We Learnt in 20 Years of L4 Microkernels?" In: *ACM Symposium on Operating Systems Principles*. Farmington, PA, USA, Nov. 2013, pp. 133–150.

[8] Norman Feske: "A Case Study on the Cost and Benefit of Dynamic RPC Marshalling for Low-level System Components". In: *SIGOPS Oper. Syst. Rev.* 41.4 (July 2007), pp. 40–48. ISSN: 0163-5980.

[9] Claudio Foellmi: "OS Development in Rust". MA thesis. ETH Zurich, 2016.

[10] *Fuchsia FIDL tutorial*.
Web: https://fuchsia.googlesource.com/docs/+/ea2fce2874556205204d3ef70c60e25074dc7ffd/development/languages/fidl/tutorial.md (visited on 04/03/2019).

[11] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter: "The Performance of \$$\mu$\$-kernel-based Systems". In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP '97. Saint Malo, France: ACM, 1997, pp. 66–77. ISBN: 0-89791-916-5.

[12] Sebastian Humenda: "Making Rust Talk. Rust And IPC On L4Re". May 2018.

[13] Sebastian Humenda: "Rust on L4Re. Integrating A Modern Systems Language Into A Microkernel Userland". June 2017.

[14] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen: "Session Types for Rust". In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. WGP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 13–22. ISBN: 978-1-4503-3810-3.

[15] Daniel Keep: *The Little Book of Rust Macros*. Web: `https://github.com/DanielKeep/tlborm/tree/5d9828b0cb5d8a1a2358f0248f4c33bd224b16f8/text` (visited on 11/20/2018).

[16] Steve Klabnik and Carol Nichols: *The Rust Language*. San Francisco: No Starch, June 2018. ISBN: 978-1-59327-828-1.

[17] *L4Re — Flex pages*. Web: `http://www.l4re.de/doc/group__l4__fpage__api.html` (visited on 02/12/2019).

[18] *L4Re Dataspace Documentation*. Web: `https://l4re.org/doc/l4re_concepts_ds_rm.html` (visited on 10/24/2018).

[19] *L4Re Interface Definition Language*. Web: `https://l4re.org/doc/l4_cxx_ipc_iface.html` (visited on 04/02/2019).

[20] Adam Lackorzynski and Alexander Warg: "Taming Subsystems: Capabilities As Universal Resource Access Control in L4". In: *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*. IIES '09. Nuremburg, Germany: ACM, 2009, pp. 25–30. ISBN: 978-1-60558-464-5.

[21] Jochen Liedtke: "Improving IPC by Kernel Design". In: *SIGOPS Oper. Syst. Rev.* 27.5 (Dec. 1993), pp. 175–188. ISSN: 0163-5980.

[22] Jochen Liedtke: "On Micro-kernel Construction". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 237–250. ISBN: 0-89791-715-4.

[23] Alex Light: "Reenix. Implementing a Unix-Like Operating System in Rust". In: (Apr. 2015).

[24] Yi Lin, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish: "Rust As a Language for High Performance GC Implementation". In: *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2016. Santa Barbara, CA, USA: ACM, 2016, pp. 89–98. ISBN: 978-1-4503-4317-6.

[25] Nicholas D. Matsakis and Felix S. Klock II: "The Rust Language". In: *Ada Lett.* 34.3 (Oct. 2014), pp. 103–104. ISSN: 1094-3641.

[26] Daniel Müller: "Evaluation of the Go Programming Language and Runtime for L4Re". MA thesis. Dresden: Technical University of Dresden, June 2012.

[27] Philipp Oppermann: *Integration Tests. Writing an OS in Rust*. Web: `https://os.phil-opp.com/integration-tests/` (visited on 11/16/2018).

[28] Gabriele Paoloni: *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. Intel Coperation, Sept. 2010.
Web: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[29] Corey Richardson: *Building seL4 Applications with Rust*.
Web: https://robigalia.org/rbg-app-2016-04.pdf (visited on 04/21/2017).

[30] Corey Richardson: *Robigalia. Rust, seL4, and Persistent Caps*.
Web: https://robigalia.org/sel4-ws-2016-12-15.pdf (visited on 01/10/2019).

[31] *SeL4 – CAmkES*.
Web: https://sel4.systems/Info/CAmkES/About.pml (visited on 02/17/2019).

[32] *TARPC, An RPC framework for Rust with a focus on ease of use*.
Web: https://github.com/google/tarpc/blob/06544faa5a0872d4be989451afc0a2b1e1278df4/README.md (visited on 03/13/2019).

[33] The Redox Project: *The Redox Operating System*.
Web: https://doc.redox-os.org/book/design/scheme/schemes.html (visited on 02/13/2019).

[34] The Rust Developers: *The Cargo Book*.
Web: https://doc.rust-lang.org/cargo/reference/ (visited on 03/01/2019).

[35] The Rust Developers: *The Rustonomicon. The Dark Arts of Advanced and Unsafe Rust Programming*.
Web: https://doc.rust-lang.org/nomicon/ (visited on 02/10/2019).

[36] Alexander Warg and Adam Lackorzynski: “Rounding pointers - Type safe capabilities with C++ meta programming”. In: (Oct. 2011).

[37] Huon Wilson: *Peeking inside Trait Objects*. Jan. 2015.
Web: http://huonw.github.io/blog/2015/01/peeking-inside-trait-objects/ (visited on 11/30/2018).

B Glossary

BID The GNU-Make based buildsystem for L4Re; builds both kernel and L4Re services, as well as external packages.

crate Rust project unit, usually a term for a Rust library, but also less frequently for binary projects.

fat pointer Pointer larger than the actual machine-specific pointer type, containing additional information such as the vtable pointer, etc. Examples are trait objects and the Box type..

message tag An additional machine word attached to each L4Re message to inform the receiver of a message about the protocol being used, the number of typed and untyped words sent and additional flags. The kernel uses the information about the number of words to copy the required data..

opcode Operation codes, used to identify the operation to be executed. It selects the interface method in an IPC operation..

ABI Application Binary Interface.

API Application Programming Interface.

AST Abstract Syntax Tree.

DSL Domain-Specific Language.

FFI Foreign Function Interface.

IDL Interface Definition Language.

IPC Inter-Process Communication.

RPC Remote Procedure Call.

UTCB User-Space Thread Control Block.

vtable virtual method table.