Großer Beleg

Evaluation of the Go Programming Language and Runtime for L4Re

Daniel Müller

Tuesday 12th June, 2012

Technische Universität Dresden
Faculty of Computer Science
Institute of Systems Architecture
Chair of Operating Systems

Supervising professor: Prof. Dr. rer. nat. Hermann Härtig
Supervisors: Dipl.-Inf. Björn Döbel
Dipl.-Inf. Michael Roitzsch

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 12. Juni 2012

____

Daniel Müller

The Go programming language was developed by Google for the purpose of systems programming making it potentially suitable for usage in microkernel environments like L4/Fiasco.OC and L4Re.
This work is meant to investigate whether Go can be used to develop services for a microkernel and to examine if there are differences in component performance and development speed in comparison to traditional development using C/C++. The investigation should focus on whether and how the concepts of Go channels and Go routines can be used for communication between tasks/processes. The implementation should be generic, but also incorporate platform specific details (for instance L4 capabilities and capability mapping).
For the evaluation, an L4 service should be implemented in Go and compared to an existing implementation.

Contents

1 Introduction 1

1.1 Motivation . . . 1
1.2 Document Structure . . . 2

2 State of the Art 3

2.1 Go . . . 3
2.1.1 Go Routines . . . 3
2.1.2 Go Channels . . . 3
2.1.3 Example . . . 4
2.1.4 Implementations . . . 5
2.1.5 Packages . . . 5
2.1.5.1 Reflect . . . 5
2.1.5.2 Netchan . . . 6
2.1.5.3 Gob . . . 6
2.2 L4 . . . 7
2.2.1 Kernel Objects . . . 7
2.2.2 IPC Gates . . . 7
2.2.3 IPC . . . 8
2.2.4 Capabilities . . . 9
2.3 Go on L4 . . . 10
2.3.1 BID . . . 10
2.3.2 Building Go Programs . . . 10
2.3.3 Language Interaction . . . 11
2.4 Related Work . . . 12

3 Design & Implementation 15

3.1 Kernel Objects . . . 15
3.1.1 Go Objects . . . 15
3.1.2 IPC Gates . . . 16
3.1.2.1 Command Thread . . . 17
3.1.2.2 Send . . . 18
3.1.2.3 Receive . . . 19
3.1.2.4 Reply . . . 20
3.1.2.5 SendIpcGate . . . 21
3.1.2.6 RecvIpcGate . . . 22
3.1.2.7 Capabilities . . . 22
3.1.3 IRQs . . . 23

3.1.3.1 SendIrq 24
3.1.3.2 RecvIrq 24
3.2 L4Go Channels 25
3.2.1 Requirements 25
3.2.2 Preconditions 26
3.2.3 Usage of Netchan 27
3.2.4 Design 28
3.2.4.1 The Interface 29
3.2.4.2 Possible Implementations 29
3.2.4.3 Proxy Channels 31
3.2.5 Implementation 32
3.2.5.1 IPC Channels 32
3.2.5.2 Multiplexing of Channels 34
3.2.5.3 Shared Memory Channels 35
3.2.6 Runtime Library Modifications 36
3.2.6.1 Go Channel Implementation 37
3.2.6.2 Hooks 38
3.2.6.3 No Hook Versions 39
3.3 Keyboard Driver 40
3.3.1 Driver 40
3.3.2 Client 41
3.4 Problems 42
3.4.1 IPC Cancelation 42
3.4.2 Initial Memory Allocation 44

4 Evaluation 45
4.1 Performance 45
4.1.1 IPC Performance 45
4.1.2 Channel Performance 47
4.2 Code Size 49

5 Conclusion & Outlook 51
5.1 Future Work 51
5.2 Conclusion 53

Glossary 55

Bibliography 57

List of Figures

3.1 A command thread can receive commands from numerous Go routines 18
3.2 Send process of a SendIpcGate 19
3.3 Receive process of a RecvIpcGate 20
3.4 Proxy channel approach for L4Go channel implementation 31
3.5 Multiplexing multiple Go Channels on one L4 IPC channel 34
3.6 Each Go Channel has its corresponding L4 IPC channel 34
3.7 An L4Go channel implementation that may suffer from a race condition 36
3.8 Nesting of multiple IPC call operations 43
4.1 Native L4 IPC and Go IPC ping-pong performance 46
4.2 Go channel and L4Go channel performance 47

List of Tables

3.1 Possible approaches for an L4Go channel implementation 30
3.2 Three hooks added to the libgo runtime library 39
4.1 Lines of code of keyboard driver written in Go 50
4.2 Lines of code of keyboard driver written in C++ 50

List of Code Listings

2.1 A sample Go program 4
3.1 Client, Go routine 1 33
3.2 Client, Go routine 2 33
3.3 Server 33
3.4 A simple Go keyboard client 41

1 Introduction

1.1 Motivation

Developing drivers and system services for an operating system is a difficult and error-prone task. This can be attributed to mainly three causes:

- *The low-level nature of the language that is used for creating this kind of software.*

  Although lots of high-level programming languages are publicly available, the important ones for systems development are C and C++. C and C++ provide the developer with a lot of freedom in what he can do [Li04], which can be both, a curse or a blessing—depending on the programmer's skills and the problem at hand.

- *The emergence of multi-processor and multi-core systems.*

  Developing software that makes use of all of a system's processing units is a tough job. This can be accounted to the programming models used, which are hard to grasp for programmers and do not provide suitable abstractions in order to make use of concurrency but still allow for the creation of software that is easily understandable. Therefore, many errors can occur, such as allocation and ownership problems between the various executing entities, race conditions, deadlocks and livelocks, and other synchronization issues.

- *The necessity for a good communication mechanism abstraction.*

  The points mentioned above affect many types of systems. Microkernels and microkernel environments, which are of particular interest for this work, pose another difficulty over monolithic ones: Due to their highly compartmentalized nature they are in increased need for communication between components, i.e., services and other processes. Establishing a convenient communication mechanism, which can be utilized easily for various communication tasks, however, is hard and it is difficult to provide an abstraction that suites all needs.

Go is a high-level language that tries to approach the aspects mentioned above. It is a safe language in the sense that it provides strict type and memory safety. In addition to that, it takes care of deallocation of no longer referenced objects by incorporation of a Garbage Collector and, thus, releases the programmer from the job of explicit memory management. For communication and concurrency, it provides easy-to-use primitives that allow development of concise and easily understandable software: Go channels and Go routines, respectively.

In this thesis, I integrate Go into L4Re in order to make it suitable for developing services and drivers for L4/Fiasco.OC in a more comfortable and safe way than is possible with traditionally used systems programming languages like C and C++.

Furthermore, I will investigate how certain L4 primitives can be integrated into the language in a user-friendly fashion, as well as how Go channels can be adapted to be useable for inter-process communication.

As an aside, it should be noted that not all of the previously mentioned points can be solved by the choice of the programming language alone. But it is—in addition to the system for which to develop itself—the major player to tackle these issues. As a system is already predetermined for my work, I will focus on the language part and leave other factors open for future work.

1.2 Document Structure

This work is structured as follows: Chapter 2 gives an overview about the fundamentals that are useful to understand this thesis. In there, I introduce the Go programming language, its basic principles and primitives. Subsequently, I cover the L4 microkernel and its L4 Runtime Environment that provides important services and abstractions, as well as the state of the integration of Go into L4Re in order to clarify which functionality I can rely upon. The last part of the chapter will compare this work to other work done in this field.

The main chapter, chapter 3, exlains the actual integration into L4Re—describing what was implemented and how it was designed. I first explain the integration of L4 primitives, so called kernel objects, into Go and will then describe how the utilization of Go channels for inter-process communication was achieved. After that, a sample service which makes use of the adapted Go channels—a keyboard driver—is depicted.

I evaluate my implementation in chapter 4. First, I analyze the speed of my developed Go IPC gate wrappers and compare it to native L4 IPC performance. The performance of my L4Go channel implementation will be evaluated as well. Further, I compare the developed Go driver to a functional equivalent implementation using C++ in terms of lines of code.

Chapter 5 concludes my thesis by providing a short summary of my work and provides an outlook for future work to be done on various related topics.

2 State of the Art

This chapter explains the basics that are necessary to understand this thesis. The first section covers relevant details of Go and gives a brief introduction into the language in general. Section two is about L4, describing the environment in which the work is done. The third section covers my previous work with regard to Go and L4, explaining some of the porting steps necessary in order to run Go programs on L4. The last part considers related work on communication constructs, the integration of systems development languages into existing environments, as well as the other direction, i.e., the wrapping of system specific features to utilize them for the programming language of interest.

2.1 Go

The Go programming language [Inc09d] was invented in 2009 by a team of three engineers—Robert Griesemer, Rob Pike, and Ken Thompson—at Google [Wik11]. It is a compiled and strictly type-safe language that is to be used for systems programming [Inc09c, Inc09d]. It has built-in support for concurrency and provides primitives for synchronization and communication. For comfortable and automated memory management, it also contains a Garbage Collector.

2.1.1 Go Routines

Go routines are the means for providing concurrency. They aim to be very lightweight such that large amounts of them can be created in order to make excessive use of concurrency. This is achieved by making them userland threads with an initially small and dynamically growing stack, that are entirely managed by the Go runtime. The runtime library also decides which Go routines to run and multiplexes them to actual operating system threads.

Go routines are created using the `go` keyword and can execute arbitrary Go functions. The executed function can have a return value, however, it is not possible to access this return value by normal means, i.e., it cannot be assigned to a variable. This would result in a syntactic error detected by the compiler. In order to pass something from within a Go routine a *Go channel* can be used.

2.1.2 Go Channels

Go channels (hereafter often referred to simply as *channels*) are communication and synchronization primitives provided by Go. They are direct language primitives, i.e., not objects implemented in a library on top of other Go constructs. Channels can be assigned to variables, passed as arguments to functions and returned from them, as well as be created dynamically at runtime. This makes them first-class objects in Go [Sco09].

Go channels are meant to be used for data exchange between different Go routines—data can be *sent* and *received* from a channel. It is important to note, however, that there is no notion of inter-process communication for Go channels. Every communication is local to the process the channel is created in.

Go channels also serve as synchronization primitives. They provide storage for a fixed amount of data items—a value that is specified during creation and cannot be changed afterwards. The default buffersize is zero, meaning the sender will block until the communication partner directly receives the value (so there is, in essence, no additional buffering involved). A channel with a buffersize of two would allow for two items to be put into the channel without blocking, and block on insertion of the third (provided the receiver did not remove a previously inserted item).

Channels are parameterized by the type they are able to transport and are strongly typed, i.e., a `chan int` can be used to send and receive integer values—not other ones—and is distinct from, for instance, a `chan string`. This way the compiler will detect incompatible type usage during the compilation phase. Henceforth, when explaining parts of the send and receive process of a channel, I will frequently refer to the transported objects of the channel's element type. For this purpose, I dub them channel items, or simply items for short.

2.1.3 Example

```
1  package main
2
3  import "fmt"
4
5  func print(c chan int) {
6    for {
7      i := <-c
8      fmt.Println(i)
9    }
10 }
11
12 func main() {
13   c := make(chan int, 0)
14
15   go print(c)
16
17   for i := 0; i < 10; i++ {
18     c <-i
19   }
20 }
```

Listing 2.1: A sample Go program

Listing 2.1 shows a sample Go program. It prints the numbers 0 to 9 on the screen and serves for the purpose of illustrating the features described above.

Every Go program (in contrast to a Go package, which contains code but is not directly compiled into an executable) must reside within the package `main` (see section 2.1.5 for a short introduction on Go packages). Its `main()` function is called on program start-up. After the

package declaration, other packages that are needed can be imported. In this case this is the `fmt` package that can be used for formated input and output [Inc09e].

The first executed statement in the shown program is the creation of a Go channel, `c`, that can be used to transfer integer values (line 13). It has no buffer space, denoted by the value 0 passed to the `make()` function. The next line starts a new Go routine using the `go` keyword. This Go routine will execute the `print()` function that will loop to receive a value from the channel which was passed in and assign it to the variable `i`. This newly started Go routine will block for receiving a value from the channel until the `main()` function reaches line 18 in parallel, where such a value is inserted into `c`. The `fmt` package is then used to print the number on the screen (line 8).

This sample also illustrates the syntax used for working with channels: sending a value is achieved using an arrow directed from the value to send to the channel to send it through (line 18). Receiving a value works the other way around—with an arrow that is directed from the channel to receive from, to the variable to assign this value to (line 7). Henceforth, I will refer to this special syntax as *arrow syntax*.

2.1.4 Implementations

At the time of this writing there exist two compilers (including runtime libraries) for Go.

Gc is the native compiler developed by the team at Google, and can be seen as reference implementation. It is actually made up of three compilers—5g, 6g, and 8g—for the different supported architectures (ARM, AMD64, and x86, respectively). This whole toolchain is derived from the Plan9 compiler suite [Tho] used for the Plan9 operating system [Lab].

Gccgo is a front-end for the GNU Compiler Collection (GCC)[1] [Tea]. As it is not developed and maintained by the team at Google, new language features generally take a bit longer to be implemented for it and the runtime library may not have the latest updates included.

2.1.5 Packages

For modularization, Go provides packages. Packages can be used to form logical units of code that belong together. These units can be imported from within a program and exported functionality can then be used.[2]

The Go standard library includes several packages in its standard distribution. Three of them are of particular interest for this work and will be discussed in more detail here.

2.1.5.1 Reflect

The *reflect* package [Inc09j] provides the means for Go programs to make use of reflection. Reflection is used for inspecting types and objects at runtime, modifying the latter, executing actions on them, or creating new ones.

In the context of this work, reflection is mainly used because Go does not provide the means for creating compile-time generic algorithms, i.e., ones that work on objects of various types—but still provide type safety. This means that in order to achieve compile-time genericity, one

[1] Gccgo is part of the official GCC project since version 4.6.

[2] Go types and functions will be exported, i.e., be visible and accessible from outside a package, if their name begins with an uppercase letter. If it is lowercase, it will be private to the package.

can pass objects of different types to a function or method only by using a common super-type and inferring the actual type at runtime—using reflection or type casts. This contrasts to, for instance, Java or C++ which offer generics or templates, respectively,

2.1.5.2 Netchan

The Go standard library also includes a package called *netchan* [Inc09h], which implements type-safe channels over a network interface. By using the loopback device, direct communication between processes is also possible.

In the netchan package, channels can be *exported* by specifying a unique name. It is then possible to *import* the channel by supplying the corresponding name. After that, values written to the channel on the one side can be read from it on the other side, just as in the process-local case.

The package is implemented using a proxy approach, where data put into the channel is transparently read from it in the background, transfered over the network to the remote process on which a listener waits for incoming data and puts it into the local channel where it can be received by the user program using the ordinary channel syntax. This way it is possible to transparently use netchan channels in places where a “normal” channel is expected.

The netchan package is implemented independent from the underlying platform and compiler—it does not use runtime internal datastructures or functions for providing its functionality, i.e., the code is portable between gccgo and gc. All functionality it relies on is already provided in the form of other Go packages.

2.1.5.3 Gob

One package that netchan depends on is gob [Inc09f]. It offers functionality for encoding Go objects into binary streams of data and decoding them back into object form. The encoder adds meta-information about the type and the structure of the object to be sent, for instance, the order and types of the attributes of a `struct` type.[3] The decoder checks this information. This way it is possible to ensure type safety, although not statically at compile time, but dynamically at runtime. However, there is no error checking performed on the actual data, i.e., no checksums are calculated and checked, and transmission errors like bit-flips would remain undetected. In the case of netchan, this is no problem, because it uses the Transmission Control Protocol (TCP) which has built-in error detection.

The encoder and decoder can handle nearly all Go types, including recursive ones such as ordinary linked lists. Types that contain cyclic references, however, e.g., circularly linked list, cannot be encoded in a generic fashion. One needs to provide special methods for encoding and decoding objects of the such types.

[3] As in the C programming language, the `struct` keyword is used to created structured types, i.e., types that embed a certain amount of objects of various types into one object of the newly defined type.

2.2 L4

As this work includes the execution of Go programs on L4Re, this section covers the relevant details of the microkernel environment and explains the terminology briefly.

L4/Fiasco.OC (henceforth referred to as *L4*) is the name of a microkernel developed at Technische Universität Dresden. It is a member of the L4 microkernel family, because it implements an Application Binary Interface (ABI) that has been derived from [Lie96]. Microkernels have the characteristic of having only security and performance-critical parts implemented in the kernel and hence running in privileged mode—everything else is running in userland without direct access to hardware or privileged instructions. In user mode, the functionality is typically provided in the form of a service—a task[4] known to other tasks which they can communicate with in order to make use of its functionality. This makes the actual kernel part slim in comparison to other kernel types like monolithic ones, which renders microkernels more suitable for security critical applications, as less code in general also means less code to be possibly exploitable.

L4 comes with several libraries offering abstractions and providing services in userland, the L4 Runtime Environment (L4Re) [Gro10]. Examples include l4re_vfs for virtual filesystem support, l4re_c wrapping L4Re functionality in a C interface for easier interoperability with other programming languages, as well as adapted versions of µclibc and libstdc++—implementations of the standard libraries for C and C++, respectively.

2.2.1 Kernel Objects

The kernel provides so called kernel objects. Kernel objects are the basic building blocks for creating programs and libraries on L4. These objects are protected by the kernel from unauthorized access or modification, which means that no userspace task can tamper with them in a way the kernel prohibits.

In order to utilize these objects, the user issues a system call—a special operation transfering control from userspace into kernelspace. The action which the kernel then performs depends on the type of object the system call is issued on and the parameters supplied.

2.2.2 IPC Gates

One of these kernel objects is an IPC gate. IPC gates provide the means for communication with other tasks—sending and receiving of messages and data. They represent one endpoint of such a messaging action. In order for an Inter Process Communication (IPC) to take place, the IPC gate has to be bound to a thread, which can then be used for receiving incoming calls. An IPC send action on the other hand, needs an IPC gate as its target object—the one that will be called, i.e., that is the destination of the send process.

Because it is necessary to have an IPC gate bound to a thread in order to be able to execute a receive operation, but every IPC gate can only be bound to one thread in the system, L4 IPC cannot be called truly bidirectional—one can always do a send, if a valid destination, i.e., an IPC gate, is known, but can only receive on an IPC gate that is bound to a thread within that task.

[4] Task is the L4 term of what is in general known as a process—an entity more or less isolated from other tasks that has its own address and object space.

In order to have bidirectional communication, two IPC gates are necessary—one for receiving in the first thread involved in the communication, and another for receiving in the second.

2.2.3 IPC

L4 distinguishes between several IPC operations that can be classified as send and receive actions, respectively. They are special flavors of the normal IPC system call that can be issued on IPC gates. L4 IPC is synchronous. In order for such a communication to take place, both partners—the sender and the receiver—have to rendezvous, i.e., both be ready for the corresponding action to take place. In order to signal that one part is ready, it does the appropriate system call which will block the invoker until the other part is ready as well.

In addition to that, it is possible to specifiy a timeout, after whose expiration (without a successful rendezvous) the whole IPC is canceled. This failure is signaled by a special return value.

The following IPC operations are available:

Call: A call can be used to send data to a given receiver and wait for a reply. The reply may contain data, but might also be just the information that the receiver has finished handling the request. Until the reply arrives, the invoker of the call is blocked, i.e., is trapped in the kernel and will not execute any code.

Send: In contrast to a call, a send cannot be replied—it only sends data to the specified receiver and cannot be used to return data. Hence, it does not provide a second synchronization point telling when the receiver has finished processing the data.

Wait: A wait is a form of open receive, i.e., one where the receiver (the one invoking the wait) does not specify from whom to receive data. The communication can be established with anyone sending data.

Receive: The receive on the other hand is given a certain sender with whom to communicate. Any sender different from the one specified will be ignored. This is also often referred to as a closed receive or closed wait.

Reply: The reply is the answer to a call on the receiver side. As mentioned before, the reply can contain additional data, e.g., the result of a computation.
A reply is a somewhat special case of IPC. It is possible only once and only in response to an incoming call.

The actual data transfer is done using the Userlevel Thread Control Block (UTCB). Every L4 thread contains a so called Thread Control Block, which is split into a kernel part, where kernel sensitive data resides, and a user part, that can freely be touched by the task the thread belongs to. The UTCB, i.e., the part in userspace, contains a fixed size memory area, called the message registers, that can be written to and whose contents (or part of it) will be copied to the corresponding message registers of the receiver of the send. The latter can then read the data.

All L4 IPC system calls work using a special *message tag* object—a bitstring specifying, among others, the amount of words, i.e., message registers, that are sent or received, respectively, as well as a label field to distinguish, for example, different operations on the receiver.

2.2.4 Capabilities

As mentioned before, kernel objects are protected from unauthorized modification from userspace. Nevertheless, accessing them must be possible. This access can be achieved using a capability—a reference in userspace to the object in kernelspace—that can be passed to the corresponding system call. Capabilities are a protection mechanism that is well established. There are many flavors of capability systems. In the case of L4, it is a so called object capability system [MYS03].

In L4, a capability is a simple index into a capability table. This table, often referred to as the object space—in addition to the address space—is local to the corresponding task and contains all of the capabilities it possesses. As the kernel knows this table, it can establish the mapping from capability index to the corresponding kernel object it represents.

After creation, a task normally owns several capabilities provided by the creator and the runtime environment. These capabilities allow for example for allocating memory, communicating with other tasks and creating new kernel objects, e.g., new threads or IPC gates (see section 2.2.2 on page 7 for a discussion of IPC gates).

IPC can also be used to map capabilities to another task. Mapping in this sense refers to the process of making the kernel object referenced by this capability available to the other task. This is for example necessary in order to make a newly created IPC gate known to another task, with which further communication is intended. This type of IPC is not substantially different from "ordinary" data IPC as the basic mechanisms are the same—the sender specifies a capability to map and the receiver can receive it. However, the sender and receiver must prepare for sending and receiving of the capability, respectively. The latter must allocate a new (empty) capability slot in its local capability table and tell the kernel to place the newly received capability within that slot. In contrast to data IPC, where the message registers are used, capability IPC uses so called buffer registers—another special region within the UTCB. The number of capabilities to map, i.e., items in the buffer registers, is also encoded into the message tag object mentioned before.

2.3 Go on L4

To run Go programs on L4, several adjustments to Go and to L4 were needed. During my work as a student assistant, I performed the necessary changes in order to run Go programs—starting with a simple *hello world* program and, based on that, implementing more complex programs. This section will summarize this work.

2.3.1 BID

L4 uses its own build system, the Building Infrastructure for DROPS (BID) [LA10], for the creation of executable programs and libraries. It provides the necessary means for dependency checking, rebuilding only components which depend on changed data, and linking with required libraries, as well as installing libraries and programs in the appropriate places, so that they can be referenced later.

The tool support focuses on three languages: Assembly, C, and C++. The default programs to be used for assembling, compiling, and linking are GNU Assembler (GAS), GCC, and GNU Linker (LD), respectively—the defaults on most Linux based systems. Although the system is written in a configurable way, the integration of the default tools is tight, making an exchange difficult. This is caused mostly by options specific to these programs being passed as arguments and not being changeable easily. Another problem is that parts of the L4 source code rely on specific GCC extensions and special LD linker scripts are used for laying out the resulting binaries.

Due to this tight coupling of GCC with BID, the first decision with regard to Go was to use the gccgo implementation, for it is mostly compatible with the GCC—easing the addition of support for Go within BID.

2.3.2 Building Go Programs

The first step of the porting was to build the libgo runtime library for L4. The libgo belongs to the gccgo project and is integrated into its build process. Both use the GNU Build System. The GNU Build System differs from BID in many ways and compiling all the source code files using BID would be a major job, as there are a lot of pre-processing steps involved, that cannot be handled easily and in a generic manner. This would also mean a lot of maintanance effort when backporting new versions of the libgo library. Due to these points, the decision was made to reuse the existing build infrastructure of libgo instead of replacing it with an L4 specific one.

When trying to build libgo it soon became obvious that a part of it, the Garbage Collector (GC), was not compatible with L4 in its current form—it uses POSIX signals for interrupting the running Go routines and making them execute code to identify no longer needed objects for later cleanup. L4, however, has only very limited support for signals by using the µclibc signal backend.[5] As using this backend would also decrease the number of supported L4 applications, a native implementation was wanted. For this, the part of transfering control to another routine

[5] As a remark: Due to its nature of being a microkernel environment, L4Re has split the monolithic libc implementation into a main part and various backends in the form of libraries that can be linked to applications individually, in order to get support for a variety of tasks, including support for signals, files, and sockets.

and back as well as the necessary synchronization had to be implemented using L4 specific thread manipulation functions and IPC primitives.

A major problem was, that signals already included the necessary work for saving the state of a thread and restoring it later—this had to be written by hand. One pitfall thereby was the UTCB. The UTCB is thread specific state, because it contains modifyable message registers and is also used for storing the return value of an IPC operation. As a consequence, it needs to be saved before interruption and restored later, which is quite costly, due to its size.

All these L4 specific changes made to the libgo are maintained in a separate version for L4Re by the operating system group at TUD.

The next step in order to build programs and packages from Go sources, was to add support for .go files to BID. This involved the modification of one Makefile directly used by BID to add rules to create object files from Go sources and link them into a package or an executable, respectively. The rest of the necessary definitions, e.g., the commands used for compiling and linking, could be written outside BID and are only referenced from the target Makefiles, defining the source files and targets to create.

2.3.3 Language Interaction

In order to write Go programs that access L4 specific functionality, it is necessary to call code written for L4 from within Go. Most of the L4 parts are written in C or C++. C is often considered the least common denominator for interaction between languages: many of them provide the means for accessing C code. Examples include Java with its Java Native Interface (JNI) [Lia99], Perl with Inline::C [Ing02], and C# with the NAG C library [Gro]. C++—although closely related to C—is problematic due to name mangling of functions, which is not standardized and may differ among compilers and even between different versions of the same compiler. Name mangling refers to the process of encoding the signature of a function, i.e., its parameters, return type, and other things, into the symbol provided in the binary. This is necessary to allow function overloading—having two functions with the same name but accepting different parameters.

Gccgo allows for accessing C functions from within Go by declaring a Go function with the corresponding signature, i.e., one where the parameters have equal sizes, and annotating it with a special "import" directive: `__asm__("function")` to state that it is defined elsewhere. Gc takes another approach and uses special cgo files that are a mixture of C and Go code and can be automatically translated into pure Go code [Ger11, Inc09a]. In both cases the opposite direction—calling Go code from within C—is not supported.

2.4 Related Work

A comparable work of porting a programming language to L4 was written by Aaron Pohle [Poh08]. In his undergraduate thesis, he developed a shell for the L4 Environment (L4Env) [Gro03, Kau06]. This work included the porting of the Python scripting language to the—now outdated—L4Env, which was later replaced by L4Re. Python is used as the glue language within the shell, that can be utilized, for instance, to chain commands. Pohle extended the language by a library that allows for comfortable communication between tasks from within the shell, and doing library and system calls. Subsequently, he evaluated the quality and speed of his Python port.

In contrast to my work, Pohle's focus was not primarily on porting a new language to L4, but to create a shell usable for administration purposes—the decision for choosing Python was made as part of his thesis and not determined beforehand. Also, his work did not involve the enhancement of language features for the underlying system. His library concentrated on wrapping existing functionality to make it easily accessible.

Another language, comparable to Go in that it contains strongly typed communication channels and lightweight threads, is Concurrent ML (CML) [Rep91], an extension to Standard ML [MTM97].

In CML channels are synchronous—the sender will block until the receiver has actually received the value sent. Built on top of channels are first class operations called *event values*. Event values help in designing more complex communication structures based on channels, e.g., messages containing an acknowledgment, signaling the cancelation of a request, or messages used for synchronization or the choice between multiple communications.

New threads can be started using the `spawn` keyword. However, there is no real concurrency on multi-processor or multi-core systems in CML—everything is executed by a single thread. An extension to this language, Parallel CML [RRX09], fixes this drawback by adding true support for multithreading.

The work on CML involved the design of a whole new language which focuses on providing powerful yet easy to use communication primitives. This differs from my work, where a language is already given and an existing communication mechanism in it is to be enhanced to make it more suitable for development on the underlying system. Whereas for CML the designer had all freedom for the implementation, I had to keep several requirements and preconditions in mind, limiting the size of the design space for possible solutions significantly.

Work in the context of writing low-level software in a high-level language was done by Madhavapeddy et al., as well as Wirth and Gutknecht.

Madhavapeddy's work [MHD+07] examines the usage of a domain-specific language, the Meta Packet Language (MPL), for the implementation of various internet protocols, with the main goal being the creation of software that has less errors compared to an implementation in an unsafe language—without a great performance penalty. It included implementations for low-level protocols such as Ethernet, IPv4, ICMP, and TCP, as well as for high-level ones like SSH and DNS. In addition to that, entire servers for the latter two protocols were created using their Melange framework, which was developed in the high-level language Objective

CAML (OCaml)—another ML based language—and compared in terms of performance and correctness to existing ones: OpenSSH and BIND, respectively.

The main difference to my work is the motivation—whereas Madhavapeddy focuses on minimizing errors in a complete software stack and even verifies this formally, my goal was to create something that is easy to use and yet does not pose unnecessary restrictions with respect to other implementations. Additionally, MPL is specifically designed for the implementation of the protocols and cannot be used for many other tasks, while Go is a general purpose language usable in many scenarios. Parallels to my work can be found in the creation of sample programs and services that are used for the final evaluation.

Wirth and Gutknecht from ETH Zurich used the high-level programming language Oberon [RW92, Wir88] to implement the Oberon operating system [Rei91, WG92]. The programming language is largely influenced by Pascal and Modula-2, incorporating strict type-safety, object-orientation, and mechanisms for good structuring. It is very simple with respect to its features and is easily implementable for compiler writers. One of the goals of the work was, again, the reduction of errors in the operating system by using a safe language and detecting errors at the earliest time possible.

Their work covers the creation of a whole new operating system as well as the design of a language for this task. I, on the other hand, did neither design a language nor write an entire operating system from scratch, but rather adapted a language for use on an existing operating system. However, as Oberon not only refers to the kernel, programs and libraries were also developed—which is one of the goals of my thesis as well.

Another work with regard to integrating system specific features into the programming language of interest for comfortable usage was done by Warg and Lackorzynski [WL11].

The paper covers the integration of capabilities of the L4/Fiasco.OC microkernel into the C++ programming language. Their approach uses operator overloading and template metaprogramming techniques to allow for intuitive utilization of capabilities in a way comparable to smart pointers [Sut02, Ale01]. The focus lies on introducing as little overhead as possible, with no additional level of indirection, no overhead with respect to pointers (as another way of referencing objects in the task-local case), and no additional dynamic memory allocations. They also introduced special versions of static and dynamic casts, taking advantage of the C++ type system.

Contrary to my work, their goal was the creation of a suitable representation of an L4 system functionality in a language, whereas for my thesis a language feature was given and its implementation had to be adjusted to utilize the underlying feature in a convenient way. My implementation is also not concentrated primarily on low overhead and high performance, but rather on keeping the changes to the runtime library minimal, in order to minimize the maintenance effort and increase the chance for integration into the main development line.

3 Design & Implementation

This chapter explains design and implementation of inter-process Go channels, presents a driver written in Go, and discusses important design decisions made during the development.

First, the wrapping of important kernel objects is described in section 3.1. Using these objects the implementation of L4Go channels (section 3.2) and an L4 sample service (section 3.3)—a keyboard driver—are covered. Finally, section 3.4 elaborates some problems that occured during the implementation.

3.1 Kernel Objects

In order to be able to write system software for L4 in Go it is necessary to provide access to several primitives provided by the kernel, i.e., kernel objects. These kernel objects include:

- Tasks as a representation of an address space and an object space
- Threads as an abstraction for serial execution of code
- Factories for providing the means for creating new kernel objects
- Schedulers for the assignment of CPUs to threads
- IPC gates for establishing a communication channel between two tasks
- IRQs to access hardware interrupts or, in the case of virtual interrupts, to provide an asynchronous signalling mechanism

Out of this set of kernel objects, mainly two are relevant for this work: IPC gates are used to establish a communication channel, which can be used for implementing inter-process Go channels. IRQs are essential for writing drivers for hardware. The keyboard driver that is to be developed requires support for IRQs from within Go in order to get notified about key events. This section explains the integration of these two kernel objects into Go.

3.1.1 Go Objects

The first step in the process of designing a wrapper for the two kernel objects, IPC gates and IRQs, was to choose the type of wrapping. Two approaches come to mind:

- *Wrap the functional interface as provided by L4Re.*

  This approach directly reuses the interface as provided by L4Re and makes it accessible to Go programs. For this to work, the functional C interface providing access to the kernel object would need to be wrapped.

The advantage of this approach is based on its supposedly simple implementation, because there is no need to derive a new interface design as it reuses the one already provided. For people familiar with the existing interface it is both easily comprehensible and straightforward to work with. For other people, however, to whom the development on L4 is unknown, this interface could be potentially hard to grasp and of a too low level of abstraction.

- *Provide a new Go type representing the kernel object.*

  The second point covers the treatment of kernel objects as Go objects of a certain type. This basically extends the first approach because it uses the L4Re interface internally, but provides a convenient wrapper in the form of a Go object around it.
  In this case, the interface design can be more abstract and, thus, potentially be understood more easily by developers who are unfamiliar with L4 internals.

When pondering between these two points, it became clear that the first approach would not be easy to implement. This is caused by the fact that, as mentioned in section 2.2.2 on page 7, IPC gates and IRQs are always bound to an L4 thread. However, because Go code is run from within a Go routine, which can be mapped to arbitrary L4 threads and as this mapping can potentially change over time, the Go developer would somehow need to manage a separate L4 thread that is bound to the corresponding kernel object (see section 3.1.2.1 for a detailed explanation of this problem). This additional work for thread creation and management should be hidden from the user of the code. With this additional constraint, the decision was made to implement a wrapper in the style as described by point two.

The next two subsections discuss the design and implementation of the wrapper objects for IPC gates and IRQs.

3.1.2 IPC Gates

IPC gates are represented in Go by two differnt classes: *RecvIpcGate* and *SendIpcGate*. Objects of type RecvIpcGate correspond to an IPC gate that is bound to a thread within the current task and can be used for receive actions. SendIpcGates on the other hand are gates that are not bound to any thread for receiving and hence can only be used for send IPC operations.

This distinction is made because an IPC gate can only be bound to one thread in a system. This thread is then able to receive data on that gate, i.e., do an open wait or a closed wait. This difference in functionality is directly reflected in the interface of the two types.

In the native L4 implementation, an IPC gate always has a label, i.e., a number, assigned to it, under which it is known to the thread it is attached to. The label can be assigned freely by the user and is returned when doing an open wait operation. This way, it is possible to distinguish the receipt of data between several IPC gates that are bound to the same thread.

In contrast, on the level of Go there is no need for the user to decide on such a label and assign it to an IPC gate when using the two types mentioned above. This is due to the fact that each thread is associated with one IPC gate. The corresponding Go object contains a reference to that thread and always communicates with it, there can be no many to many mappings of gates to

threads, and so there is no need for distinguishing between gates from within one thread—which would be what the label is for.[1]

3.1.2.1 Command Thread

As briefly explained when dicussing the kind of wrapper to use (section 3.1.1), a special thread is needed in order to use IPC gates from Go code. This has two reasons:

- Receiving data from an IPC gate can only be done from within the thread that is bound to the gate. If one would do a receive operation from within a Go routine, however, it depends on the mapping of the Go routine to the actual L4 thread whether this operation can actually succeed. If the Go routine is run by the thread that is bound to the IPC gate, everything is fine. If, on the other hand, the Go routine executes on a different thread, the receive can never be handled—and the Go routine will be blocked for an indefinite amount of time.
- In order to send or receive data using IPC the UTCB is used. The UTCB is bound to an L4 thread. If the user wants to write data into the UTCB from within Go and the underlying L4 thread executing the Go routine changes, the UTCB will change as well. This can lead to writing data to the wrong UTCB or using the wrong one for the send i.e., not the one where the data was written to. This argumentation applies to the process of receiving data as well.

To solve these two issues, a special thread, the *command thread*, is introduced. A command thread is an L4 thread that is directly created using the L4 Application Programming Interface (API) for thread creation. This way the Go runtime does not know of its existence and does not manage it like a Go routine. This thread is the one that is bound to the actual IPC gate. It is prepared for receiving commands from Go routines and executing them. Commands in this sense represent the actual IPC operation to execute, e.g., a call or an open wait.

It is important to note that the command thread can receive commands from any Go routine and is not bound to a specific one (see figure 3.1). However, it can handle only one command at a time. Until it is finished all other senders will block. This behavior makes these Go objects easy to work with, even if they are shared between multiple Go routines, because no additional synchronization is needed. Only the UTCB, used for reading and writing data, complicates things and needs additional means of synchronization if the IPC gate object is shared between multiple Go routines—otherwise read and write conflicts might occur.

An alternative approach to coping with these two problems would be to pin the mapping of Go routines to L4 threads, at least for the time of preparing and executing an IPC operation. This would allow this operation to always be executed from the Go routine that is run on the thread which is bound to the IPC gate—a necessary precondition for the action to succeed.

However, this change would require the Go runtime library, which is responsible for mapping Go routines to operating system threads, to be adapted to support this fixing of Go routines to

[1] Internally, i.e., on the level of C++, the IPC gate is bound to the thread using a specific label, however, it can be the same for every gate because there is no need to distinguish between various gates if there is only one gate per thread.

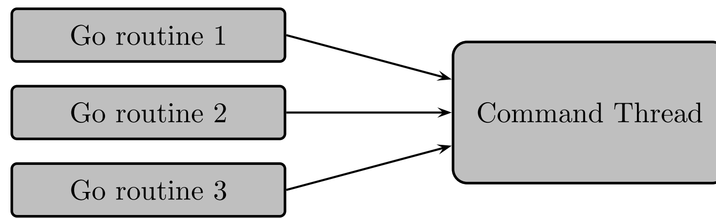

Figure 3.1: A command thread can receive commands from numerous Go routines

L4 threads. As the modifications to this library should be kept as few as possible, this approach is rejected.

3.1.2.2 Send

Using a command thread as described before, it is possible to implement the sending of data from within Go code. A detailed illustration of the sending process for communication of a client task with a server task is shown in figure 3.2 on the facing page.

The first step for the Go code is to write the data that is to be transfered into the UTCB. This can happen in a specially encoded form, having meta information attached to the data, e.g., for allowing error detection or passing type information, or simply as a bytewise copy of the memory region of interest. The behavior is entirely up to the Go code, the only constraint is that the receiver must know the format in order to decode it.

After that, a special label is created. It encodes the amount of message words to transfer, as well as the IPC operation to exercise, i.e., send or call. This label is then used for the IPC call to the command thread. The actual call does not transfer any message registers, because the data written to the UTCB by the Go routine was written directly into the UTCB of the command thread and any transfered words would overwrite this data. This is also the reason for encoding this information within the label in the first place and not using the message registers for transportation.

The command thread waits for an incoming IPC operation using an open wait. After that, it decodes the label that was transfered from the calling Go routine. Depending on the desired IPC operation (the “command” in figure 3.2) the actual IPC to the receiving task is performed, transfering the amount of message registers as encoded within the label. After it returns, the thread replies to the caller in order to unblock it. The behavior of synchronous IPC is preserved, i.e., a call blocks until the actual receiver replied. This is achieved, because the reply to the call done by the Go routine for invoking the command thread takes place after the actual IPC operation finished.

Due to the mechanism explained above it is possible to send data to any partner that can receive an IPC—there is no Go specific protocol or metadata involved. This also means that communication with non-Go tasks is possible—a requirement for, for instance, transparently replacing a service written in an unsafe language like C++ by one written in Go.

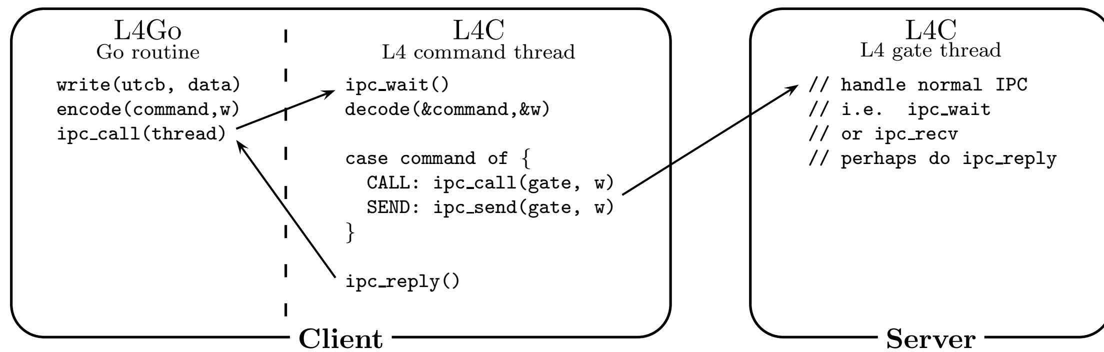

Figure 3.2: Send process of a SendIpcGate

3.1.2.3 Receive

The receipt of data from another task poses another problem over the send case, that will be tackled here.

When receiving data using the command thread, it is possible that an actual data IPC from some other task is mistakenly interpreted as a command to the thread or vice versa. This is due to the fact that the receipt of a command uses an open wait operation—because it cannot know the source of the command IPC. The actual IPC operation to execute, however, is some sort of receive as well (it does not matter right now whether it is an open or closed one). It now depends on the timing of the senders, which of the two communications is treated as the command and the data IPC, respectively—which is essentially a race condition.

In order to solve this issue, it would be necessary to make the first receive—the one for receiving a command—a closed receive, i.e., an operation which can only be served by a known sender. In that case it could not accidently receive the data IPC, because this IPC would come from a different source. Now, even if the operation to be executed by the command thread is an open wait (for a closed wait the same argumentation as above applies), it still could not get mixed up, as there can be no waiting for a new command in parallel, because everything happens within one thread, where execution is strictly serial.

To achieve this behavior, an additional thread is introduced: the *forward thread*. Just like the command thread, it is created as a native L4 thread. The forward thread is used as the fixed communication partner for the command thread—the latter can now receive commands using a closed receive that can only be served by the forward thread. Data and command communications can no longer be mixed up.

Using this forward thread and a command thread the RecvIpcGate type is implemented. Objects of this type can be used to receive data. Figure 3.3 on the next page illustrates this process.

The first step here is to encode the desired command to issue in order to send it to the command thread. As in the send case this involves the creation of a special label. After that, the forward thread is called. It decodes the label to look up the desired IPC operation. In case of an open or closed wait, a call to the command thread is performed. In case of a reply a normal send will be executed. This difference is due to the blocking properties of these IPC operations. A wait, no matter if it is open or closed, blocks until something is received. A reply on the

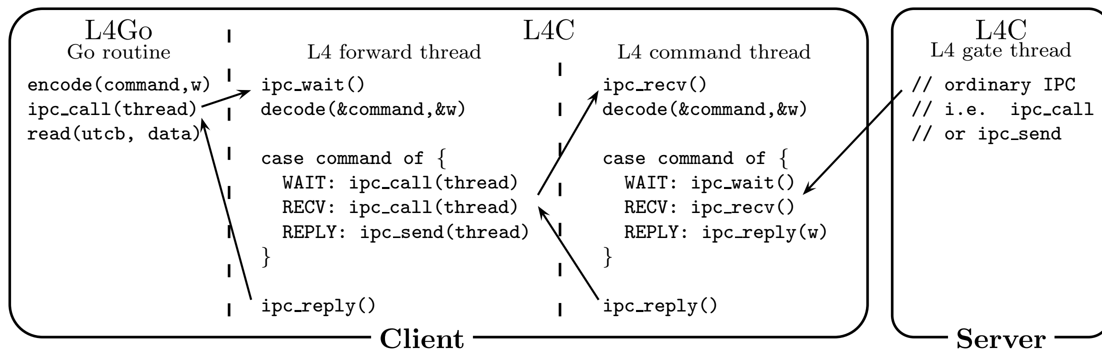

Figure 3.3: Receive process of a RecvIpcGate

other hand acts just like a send—it returns immediately. The command thread will receive the command encoded in the label. The difference to the send case is that the wait operation is closed, meaning it knows the forward thread and waits only for commands from exactly this thread. After decoding the command to execute, the corresponding operation can be invoked. As in the send case the reply signals the forward thread to unblock, which in turn replies to the caller—unblocking it as well.[2]

The Go code can now read the received data from the UTCB. In this case it will know the UTCB of the thread that did the actual receive—the command thread. The amount of received words is also passed back to the caller. It is, again, encoded within the label of the replies. This is not shown here for the sake of simplicity.

3.1.2.4 Reply

The implementation of the reply action deserves a more detailed look, although it is briefly covered implicitly above.

In order to allow for replying to an IPC call operation, the L4/Fiasco.OC kernel creates a reply capability whenever a call is received. This reply capability can be used at most once and only in order to send something back to the caller. After that usage it will be invalidated. It is not possible to preserve multiple reply capabilities, e.g., for answering multiple incoming calls—only the one from the last call will be valid. This limitation of having only one reply capability applies only in the context of one thread, i.e., each thread in the system can have its own reply capability—with respect to these capabilities these threads are isolated from each other.

This has implications for the implementation of the command thread for receiving actions as shown in figure 3.3. In order for an incoming reply command to be successfully executed, the reply capability must still be valid, which implies that there must have been no reply issued after the incoming call of interest but before the actual reply command.

[2] The process as shown here is simplified to make it easier understandable. If the command thread executed a reply operation it will not reply itself to the forward thread, because the latter did only a send operation for issuing the command.

In order to achieve this, the command thread does not use the reply capability to answer calls—it does a normal send IPC operation instead. For this to work two requirements must be met:

- *The caller must be known to the command thread.*

  This condition is fulfilled trivially, as it must be known in order to do a closed wait. This was the requirement for introducing the forward thread in the first place (see section 3.1.2.3).

- *The caller must be able to receive an incoming IPC operation.*

  In the inter-process case this is only fulfilled if an IPC gate is available. However, the command thread and the forward thread are both within the same task. And communication between threads within a task does not require a separate IPC gate—a thread can receive data from another thread within the same task simply by doing the corresponding IPC system call, i.e., a wait or a receive.

As both requirements are satisfied, it is possible to not use the reply capability to answer a call, but simply do an ordinary send operation instead. This allows the original reply capability to the remote IPC partner to be left untouched until the actual reply command is issued by the user.

3.1.2.5 SendIpcGate

With the functionality for sending data as described in the previous section, I am able to implement the SendIpcGate type.

The first step in the design of this type is to decide on a way to put the data to transfer into the UTCB and read it from there.[3] The Go standard library defines interfaces for writing data to and reading data from various sources—`io.Writer` and `io.Reader` [Inc09g], respectively. Types implementing these interfaces are used in various Go packages, e.g., fmt [Inc09e] for formated input and output, zlib [Inc09l] and zip [Inc09k] for on the fly compression and archiving of data, respectively, and opengpg [Inc09i] for encryption using GNU Privacy Guard (GPG).

By implementing the `io.Writer` and `io.Reader` interfaces, i.e., adding `Write()` and `Read()` methods with the corresponding signature, SendIpcGate objects can be used everywhere where an object implementing one of these interfaces is expected. This allows for a seamless integration of IPC gates into existing Go code.

After that first step, the actual IPC methods were added: `Send()` and `Call()`. These do not accept any parameters. They know implicitly about the UTCB where the data is written to, because they know the command thread to which the UTCB belongs. They also have knowledge about the number of message registers that contain data, as the `Write()` method records the amount of items written. With this knowledge, these IPC methods can invoke the corresponding IPC operation on the IPC gate represented by this object.

To summarize, the SendIpcGate interface contains four methods: `Write()`, `Read()`, `Send()`, and `Call()`, as well as a constructor function, doing the necessary initialization

[3] The possibility for reading data from the UTCB is needed for the SendIpcGate, because the reply to a call can be used to send data—which would be extracted by reading it.

work, like the creation of a command thread.[4] The usage involves two or three steps, depending on the IPC operation: the writing of data into the UTCB, the invocation of the desired IPC method, and, in the case of a call with a reply sending data, the reading of the answer.

3.1.2.6 RecvIpcGate

For receiving an IPC using an IPC gate, the RecvIpcGate type was created. In addition to open and closed wait operations it must also have support for replying to an incoming call.

As for the SendIpcGate type, support for both, the `io.Writer` and the `io.Reader` interfaces has to be added. Reading must be supported for the obvious reason of reading data that was received. Support for writing data to the UTCB is necessary in order to be able to send back data in the reply to a call.

The interface of the RecvIpcGate type comprises five methods: `Write()` and `Read()` as before, as well as `Wait()` for doing an open wait, `Recv()` for a closed wait, and `Reply()` for replying to a received call. As for the SendIpcGate, an additional construction method is responsible for creating the previously explained forward and command thread.

3.1.2.7 Capabilities

As explained in section 2.2.3 on page 8 that introduced L4 IPC, the mapping of capabilities is performed using IPC as well. The basic principles are the same as for "ordinary" data IPC, escpecially the need for an IPC gate for receiving a capability mapping is unchanged. Due to the fact that mapping capabilities between tasks is an essential functionality for systems development on L4, support for this feature has to be included in the RecvIpcGate and SendIpcGate types as well.

The mapping of a capability does not differ much from the sending of data—one has to specify the capability, put it into a special part of the UTCB (the buffer registers—in contrast to "ordinary" data that uses the message registers), and tell the kernel the number of capabilities to transfer. After invoking the corresponding IPC operation, the IPC partner can receive the mapping.

The receiving of the mapping includes a little additional setup. The receiver has to allocate a new capability slot, into which the kernel can insert the received capability. After informing the kernel about this slot (or slots, if one wants to receive multiple mappings) and finishing the receive IPC operation, it contains the mapped capability, which can from now on be used by this task by supplying the capability index representing it.

To add support for mapping capabilities to the RecvIpcGate and SendIpcGate types, the interface has to be enriched.

The SendIpcGate type is extended with three methods: `CallSendCap()`, `CallRecvCap()`, and `SendCap()`. The first one can be used to send a capability using a call operation, whereas the second one receives a capability in the reply to it. `SendCap()` sends the capability using an IPC send. In addition to that, analogous methods for sending

[4] Go has no notion of constructors in the traditional sense. In C++ and Java, for instance, every method that has the same name as the class it belongs to is considered to be a constructor, i.e., the method for creating objects of this type. In Go, every function can be used to construct objects—the naming is entirely up to the developer of the code.

multiple capabilities, i.e., a vector of them, are introduced for convenience and to not restrict the interface artificially to one mapping at a time: `CallSendCaps()`, `CallRecvCaps()`, and `SendCaps()`—making up a total of six added methods.

The only restriction this interface poses over the underlying native L4 implementation, is that it is not possible to send and receive a capability together in one call, i.e., send one in the send part of the call and receive one in the reply to it. A method offering this behavior could easily be created, but due to missing use cases is not yet implemented.

The RecvIpcGate type also is extended with six additional methods: `WaitCap()`, `RecvCap()`, and `ReplyCap()` for receiving a capability using an open wait, receiving one in a closed wait or sending one in a reply, respectively, and their counterparts accepting multiple capabilities.

An alternative approach for an interface would be to allow for writing capabilities to the UTCB and reading ones from it using methods similar to `Write()` and `Read()` used for data, respectively, which keep track of the next entry in the corresponding buffer to operate on automatically. This would have the advantage of being more intuitive to handle, due to the similarity in usage with respect to sending and receiving data. However, this is not easily possible due to the setup required to receive a capability, i.e., the allocation of a capability slot. In essence, before receiving, one would need to know that this receive operation is meant to be used for receiving a capability mapping along with the amount of capabilities to receive. This could be implemented with some sort of internal data exchange before the actual receive operation using a specifically designed protocol. However, as implementing this functionality is not trivial and would exceed the amount of time available for this thesis, the interface as previously described was used.

Having defined the interface, the actual implementation is straightforward. The main work is to extend the command thread for handling the transfer of items, the L4 term for a special kernel object, e.g., a capability. This includes encoding the number of items to send or receive into the label describing the action to do. On the side of Go, support for allocating capabilities needs to be added, as well as the necessary setup code to execute before a receive could happen is to be implemented.

3.1.3 IRQs

IRQs are another type of kernel object in L4. They provide a mechanism for asynchronous notification. IRQs can be seen as an abstraction for two different types of notifications:
On the one hand, they can be used to communicate with hardware that triggers hardware interrupts as an information that a special event occured, for instance, that a network package arrived at the network interface card, or that a key was pressed or released on the keyboard.
On the other hand, they might also be used without interaction with hardware, as a so called virtual interrupt, that is implemented entirely in software but serves the same purpose of providing an asynchronous notification mechanism.

Because support for this kind of notification is another essential feature for system development on L4, IRQs have to be wrapped in the form of Go types as well.

The implementation of Go IRQ objects is analogous to that of IPC gates: there are two distinct types, *SendIrq* and *RecvIrq*, representing an IRQ kernel object to send and receive notifications, respectively. Like IPC gates, IRQs have to be attached to a thread in order to receive a notifi-

cation. The other way—triggering one—is always possible without such a binding. The same constraint as before, that one IRQ can be bound to at most one thread in the system at a time, essentially led to the distinction of these two types.

3.1.3.1 SendIrq

The SendIrq type provides a method `Trigger()` for triggering such an asynchronous event. On the level of L4 this is a system call that is executed on the underlying IRQ kernel object, comparable to, for instance, send and receive on an IPC gate, for sending data or doing a receive operation, respectively. In contrast to SendIpcGate, there are no `Read()` and `Write()` methods, because there is no need to read or write data, as IRQs cannot be used to transfer messages in a way similar to IPC gates.

Like the SendIpcGate type, the implementation uses a command thread to execute the actual operation. The rationale behind this usage is the same as well: in order to trigger such an IRQ object, a fixed UTCB is needed (see section 3.1.2.1 on page 17). This fact is not obvious—as just explained, there is no data to be transfered using the UTCB—and is rooted in the implementation of the trigger system call. It is implemented as a protocol on top of L4 IPC, which uses the UTCB for passing special protocol-specific data. Using this data, the receiver can execute the actual IRQ operation.

3.1.3.2 RecvIrq

Receive is another operation that IRQs provide. It can be used to wait for a notification on a specific IRQ object. RecvIrq accounts for that operation with a corresponding `Recv()` method.

Like the SendIrq and SendIpcGate types, RecvIrq has a comparable implementation to RecvIpcGate: two additional threads are used to achieve the correct behavior. A command thread allows for the correct receipt of notifications in the first place and is needed due to the variable mapping from Go routines to L4 threads, as explained before. A forward thread is necessary in order to mask out IPC messages that are received by the command thread that are no actual commands, but used otherwise. By introducing the forward thread as a fixed communication partner and using a closed receive from this source, the command thread cannot mix up these messages with actual commands.

3.2 L4Go Channels

A major part of the work involves the creation of Go channels that can be used for inter-process data exchange—I refer to these channels in the following as *L4Go channels*, in contrast to the "ordinary" Go channels which can only operate within a task-local context. With the IPC gate primitives for Go as described in the previous section it is possible to implement these channels. This section describes their design and implementation.

First, some requirements are defined that have to be fulfilled by these channels. After that, preconditions as determined by the Go language itself are explained, which directly influence the resulting design. Next, I elaborate on the implementation and explain possible alternatives, as well as discuss necessary modifications to the Go runtime library.

3.2.1 Requirements

In order to decide on an L4Go channel design, five requirements have to be fulfilled. The reasoning leading to the final design will be explained below. These requirements are:

R1) *The changes made to libgo should be as few as possible as this directly affects the maintenance effort and possibility of integration into libgo's main development branch.*

The libgo runtime library is developed and maintained by the team responsible for the gccgo. Bug fixes and new developments will be implemented there. In order to make use of the improvements on L4, regular updates of libgo are required.
Updating libgo, however, also means adjusting the changes that were made to it previously in order to make it work on L4 again. This maintenance effort of reapplying the changes to the new version can be substantial, depending on the amount and type of adjustments the developers made. It is therefore important to keep the modifications to the library minimal, to allow for frequent updates of libgo.
Another fact to consider is the possibility of having the modifications integrated into the main branch. In that case the gccgo team would take the responsibility of maintaining the L4 specific changes and they would become part of the official distribution of libgo. Aside from the fact that the popularity of L4 would play an important role for that process to succeed—which cannot be influenced by this work—the amount of changes is relevant as well. The less changes there are, the less maintainance and test effort exists for the gccgo team and the higher are the chances of integration.

R2) *The special channel syntax for sending data through a channel and receiving data from a channel (arrow syntax) has to be preserved for the case of inter-process channels—a feature relevant for the integration into existing Go code as well as into Go in general.*

If existing Go programs were to be ported to run on L4 and want to make use of L4Go channels for distributing work between several tasks or for communication in general, adjusting them is easier if already present Go channels can be reused and need only be initialized once (to make them refer to a remote endpoint instead of a local one) instead of changing every occurrence of a channel in a syntactic way, i.e., replacing the send or receive syntax with a method call on a special object.
Preserving the send and receive syntax is also favorable for seamless integration into the

Go language in general, as users will immediately recognize channels as such—just as in the local case.

R3) *The semantics of Go channels, especially their synchronization properties, must be retained.*

Go channels cannot only be used for communication between Go routines but also for synchronization: if a channel is full, i.e., its buffer memory has no slot left for a new item, sending a new object is not possible and the corresponding request will block until an element was received from this channel, making one slot available for a new item. Analogously, receiving an object from a channel with an empty buffer will block as well. This behavior allows Go code to use Go channels as the means for various synchronization tasks, e.g., for triggering a certain action upon unblocking of a receive from a channel. As this is a basic and well specified property of Go channels, which may be relied upon by numerous existing Go applications, this behavior has to be preserved.

R4) *Channels have to support bidirectional data transfer—it must be possible to send and receive data on one and the same channel.*

"Normal" Go channels are bidirectional. If L4Go channels only allowed for unidirectional communication, this would limit the usage and restrict the number of possible applications. For example it would no longer be possible to send data on a channel and receive a result on the same one—a typical scenario at least for simple services.[5] Also, L4Go channels should integrate as seamlessly as possible with existing Go code, i.e., with as few changes as necessary.
In order to achieve that, they should also support bidirectional communication.

R5) *A special type of channel should be available, allowing the sending and receiving of capabilities.*

This fact covers the treatment of a platform specific detail—the mapping of capabilities on L4. Early in the design process the possibility for creating a special type of channel used for sending and receiving capabilities was elaborated. These channels would provide for an easy way of transfering access privileges between tasks. Usage scenarios include server setups, where a manager can map resources to clients.
The general concept of a capability channel does not differ from an ordinary L4Go channel—an object of a specific type is sent from the first task and can be received by another one—but the implementation would differ. Although mapping a capability involves IPC as well, a special setup is necessary beforehand and the actual process of sending differs as well.

3.2.2 Preconditions

In order to comprehend the design decisions made for L4Go channels, it is important to understand the preconditions that are determined by the Go programming language:

[5] More complex services would presumably use a different type for the reply than the one that was given as input—and would thus need a distinct channel for providing the answer as well.

- *Go channels are embedded in the language—not built on top of it.*

  With this point I want to emphasize the fact that it is not possible to modify channels or their behavior by white-box reuse techniques such as inheritance or overwriting of methods.
  If Go channels were ordinary language objects, one could think of inheriting from them and changing the implementation of the send and receive methods. This is not possible in this case, because channels are directly built into the language. Send and receive operations are translated by the compiler into corresponding runtime library calls. Modification of the behavior of these functions would require changes to the runtime library itself as well—no modification hooks are provided.

- *Go has no support for operator overloading.*

  This point is closely related to the first one, but not as far-reaching. It covers the possibility offered by some languages to modify the behavior of (certain) operators provided by that language. C++, for instance[6], supports overloading of operators [fITS03]. This means that the functionality of operators for user-defined types is entirely up to the author of the code.[7] All the developer has to do is write a special operator function and implement the desired behavior.
  If Go supported overloading of operators, one could keep the arrow syntax for sending and receiving on channels and would only need to provide a user-defined type, i.e., some sort of remote channel object that is connected to an L4 service, for which these operators would be overloaded. If the arrow syntax is then used, a user-defined function would be called that would carry out the corresponding IPC actions.
  Unfortunately, Go does not provide operator overloading as a language feature [Inc09b].

3.2.3 Usage of Netchan

The first possibility that was considered was the usage of the netchan package. It provides—as explained before (see section 2.1.5.2 on page 6)—remote channels on top of a network interface. There are, however, arguments against this:

- *The abstraction is inappropriate, making the interface complex and the overhead high.*

  This point refers to the fact that netchan uses network interfaces as the abstraction for the underlying communication primitive for transfering data.
  A network interface is a very high-level abstraction. This becomes obvious when examining the interface provided by netchan [Inc09h]. It consists of two types, an `Exporter` and an `Importer`. An `Exporter` is bound to a specific TCP port. It provides, among others, methods for exporting a channel under a specific name and for synchronizing. The `Importer` provides a similar interface, but for importing channels that were exported previously. Although the rationals for all the details of this interface are not entirely clear to me, an implementation that directly uses L4 IPC primitives without a network abstraction layer could be much slimmer: there is no need to distinguish between TCP ports

[6] Other languages with support for operator overloading include Eiffel, Haskell, and Smalltalk.

[7] C++ does not support overwriting of operators for built-in types, i.e., at least one operand in an overloaded operator function needs to be a user-defined type.

or an equivalent, or to multiplex multiple channels over a single one—a distinct L4 IPC channel could simply be created for each L4Go channel. The need for synchronizing data also vanishes in the case of IPC. For a network connection there is no guarantee when packages arrive, due to an unreliable network being the medium of transfer. In case of L4 IPC there is no such unreliable component in between—the kernel takes care for correct synchronization.
An additional fact is the high overhead of network transfers in contrast to IPC on L4. In order to provide network related functionality a special service is used. Additionally, there is a driver for the network interface card involved.[8] These additional components slow down the communication speed and add complexity that can be avoided by using a different communication primitive—L4 IPC, i.e., an IPC channel instead of a network connection.

- *Netchan channels are unidirectional—a direct restriction to the usage of channels.*

  Channels in netchan can only be used to send *or* receive—not both. The desired direction has to be decided upon when connecting the channel and cannot be changed later on. Normal Go channels, however, are bidirectional. This opens the doors to potential problems—for instance the wrong usage of a netchan channel, i.e., sending on a receive-only channel or receiving on a send-only channel, which will block the sender or the receiver indefinitely. This can lead to errors that are hard to detect as they only occur at runtime.
  In addition to that, this property of netchan channels is also in direct conflict with requirement R4.

- *Implementing capability channels on top of netchan is impossible.*

  This point discusses the use of netchan for providing capability channels. As explained in section 3.2.1 on page 25, a special type of channel for sending and receiving capabilities should be created in order to fulfill requirement R5. Capabilities in L4 can only be transfered (mapped or granted) to another task by using IPC. This has two implications: First, network channels are not local to one system but can connect several systems and capabilities are only valid in the context of one and the same L4 system. Due to this, a network interface, again, appears not to be the right abstraction to use. And second, as networking in L4 is only built indirectly on IPC—the communication with the network service is done using IPC—it is not possible to built capability channels using only networking facilities. One would need to break the abstraction and rely on IPC for transfering capabilities.

These three points and the synchronization problem as discussed in section 3.2.6 disqualify netchan for the implementation of L4Go channels.

3.2.4 Design

After discarding the idea of using netchan for implementing L4Go channels, a new implementation has to be designed. Therefore, an appropriate interface as well as a concept of the implementation need to be developed.

[8] In the case of computer local network traffic, i.e., traffic over the loopback device, there would be no need for a driver for a network interface card.

3.2.4.1 The Interface

The first step is to design a suitable interface to be provided for interacting with L4Go channels. In principle there are not many requirements to be fulfilled by this interface, which, thus, can be very narrow:

I1) The first requirement stems from the fact that L4 is an object capability system. It states that an L4 IPC connection to be used (at least initially) must be somehow granted to the task and cannot be established entirely by itself: as L4Go channels are to be used for communication with another task, it is necessary to have some communication channel to this task, either for direct information exchange or to be able to establish further channels by means of mapping capabilities that represent this ability to communicate. Without this initial channel no propagation of authority, i.e., right to communicate, is possible.[9]
To account for this requirement, a function is introduced to which this initial communication channel gets passed in. In the case of L4Re, this can happen as a string parameter, containing the name of the L4 IPC connection, under which it is registered in the environment and can be retrieved using the corresponding function for this job.

I2) The second requirement was already given earlier: R2 states, that the existing channel syntax has to be used.
This means that it is not possible to implement dedicated `Send()` and `Recv()` functions for use with these channels which can be used for sending and receiving data, respectively, but rather that it is necessary to find a way to preserve the usage of the arrow syntax.

In summary, it can be stated, that the interface has to contain a setup function or method, that can be used to initialize the channel and to pass in an L4 communication channel to use. By preserving the already existing arrow syntax, the remaining part of the interface comprising the sending and retrieving of data items is fixed as well.

3.2.4.2 Possible Implementations

After having decided on the basic design of the interface, a possible implementation has to be found. It is possible to integrate Go channels on various levels—in every case, however, the interface as described above has to be provided. Three basic approaches come to mind:

A1) *An implementation which is worked into the language.*

On the lowest level possible, L4Go channels could be directly embedded into the language. Go channels already are language constructs. By adapting the compiler, support for binding a Go channel to some sort of L4 IPC connection could be added, and send and receive operations on this channel could use this connection to transfer data items across task boundaries. This approach would mainly involve two steps:
First, the construction procedure for channels, i.e., the `make()` function, would need to be adjusted in order to accept an additional paramter. This argument references the IPC connection to be used, as explained in the discussion of the interface. This step satisfies

[9] The requirement of having such a channel is a feature of object capability systems, that allows for confinement, i.e., restricted authority propagation. See [MYS03] for a more detailed explanation of this property.

requirement I1.
Second, a way to change the action of the send and receive operations using the arrow syntax would have to be found, to make use of the given IPC connection for sending and receiving data items, in order to fulfill I2.

A2) *An implementation in the libgo runtime library.*

On an intermediate level, only the runtime library could be modified. In that case it would no longer be possible to adapt the built-in channel creation function, `make()`, for "connecting" the channel to a different task, because this adaptation would result in a syntactical change to the language, which would need compiler support to be implementable. In order to still fulfill I1, an additional function that can be passed in a reference to the communication channel to use, which performs the necessary initialization work, could be introduced. This would result in a two step process for setting up an L4Go channel—creating the channel using the `make()` function and binding it to the IPC channel—in contrast to "ordinary" Go channels, where only the first step is needed.
Preserving the channel syntax, however, is still possible, because the arrow syntax is directly translated into calls to functions of the runtime library by the compiler in order to execute the corresponding action. These functions would need to be adapted to use IPC for data transfer.

A3) *An approach similar to that taken by netchan, implemented as a separate Go package.*

One could also implement L4Go channels without any modifications to compiler or runtime library, but as an independent Go package. This would be the highest level possible for the implementation.
Using this approach, an additional function as depicted in the previous point is necessary in order to satisfy I1, too.
For fulfilling requirement I2, the same strategy as employed by netchan could be used. The way it is done there, is based on a proxy-object approach, as briefly described in section 2.1.5.2.

For deciding which approach to use, the different advantages and disadvantages were discovered and weighted against each other. They can be summarized as follows:

Approach	A1	A2	A3
Modification of...	compiler & runtime library (−−)	runtime library (−)	neither (++)
Development effort	−−	−	+
Integration	++	+	+
Expected performance	++	++	−

Table 3.1: Possible approaches for an L4Go channel implementation

As can be seen in table 3.1, approach A1 is pretty radical: it provides a great integration of L4Go channels into the language, due to the syntactical changes that can be introduced, especially for adaptation of the built-in `make()` function. The expected performance is also good,

because of the possibility of a direct modification of the functions for sending and receiving channel items—there is no additional level of indirection involved.

The necessary modifications to compiler as well as runtime library and, associated with this, the high development effort, however, disqualify this approach from further considerations. The main argument here is the fact that modifications to the runtime library—and even worse: the compiler—should be kept as minimal as possible.

This is also the reason why approach A2 is not used. Although it is easier and faster to implement than the previous approach, as only the runtime library needs to be adapted, this remaining modification still has more impact on the applicability of the resulting implementation with respect to maintenance effort and chance of integration into the main development line than, for instance, the points of integration or performance.

The last approach, approach A3, is the one of choice. Here, the runtime library as well as the compiler need not be changed. Additionally, the development effort is reduced compared to the two previous approaches, because the implementation could incorporate the previously implemented Go IPC gate objects and can happen nearly entirely using Go, instead of C, which would need to be used for adapting runtime library and compiler.

The expected performance, however, is not as good as in the previous cases. This is due to overhead introduced by another layer of indirection: after sending a channel item from Go, it will first be read from the channel in background, sent to the remote task using IPC means, be written transparently into the local channel there, and can then be received from within Go user code. The next section describes this process of transfering data in more detail.

3.2.4.3 Proxy Channels

As just explained, an approach similar to that taken by netchan will be used for the implementation of L4Go channels. I dub this implementation *proxy channels* or the proxy channel approach. This name refers to the fact that these channels do not actually send the data items to the remote task themself, but rather provide the interface for this sending to other code. They are necessary to preserve the arrow syntax. Conceptually, a proxy channel works like this:

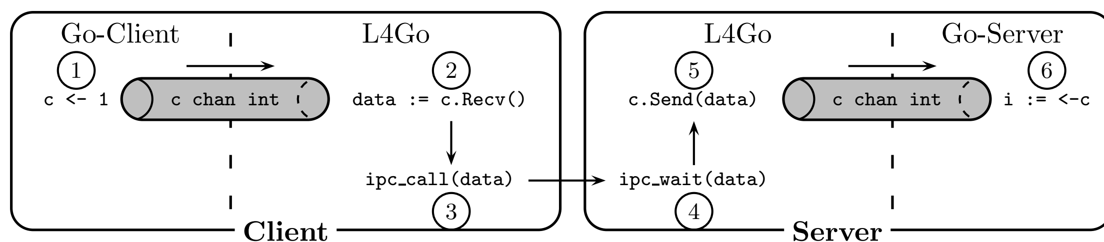

Figure 3.4: Proxy channel approach for L4Go channel implementation

The process illustrated in figure 3.4 uses two L4 tasks: a *Client*, C, and a *Server*, S. The client wants to send some data to the server, using a L4Go channel, `c`. This channel must be initialized (“connected”) to refer to the server. S also has such a channel on its side—it is connected to the client. For the following explanations, I assume that the channels are unbuffered, but it is possible to use buffered ones here as well.

The client starts by putting the data to be sent into its local channel using the channel send syntax (1). On the other end of the channel a Go routine was installed by the initialization function, which loops in an endless loop to receive data from the channel (2).[10] Each time a new data item ("data") is received from the channel, it will be sent to the receiver task, which the channel is connected to (3). This sent uses the IPC functionality as provided by L4.

The server loops as well to receive data from C using the IPC counterpart to the send operation the client uses (4). When it receives the data item, it puts this element into its local channel (5). All this happens completely transparent to the user, within an encapsulated package that was used for setting up the channel. Now, the actual transfer to the Go user code happens. At (6) the data item is read from the local channel and can be processed further.

3.2.5 Implementation

3.2.5.1 IPC Channels

When trying to implement L4Go channels that use an IPC connection for the actual data transfer from, say, a client to a server, two approaches are possible:

C1) *Use a unidirectional IPC connection from the client to the server.*

This approach uses a single IPC connection for all data transfers. For this to work, there must be some sort of agreement between both sides, i.e., the sender and the receiver, on how the communication works. As this connection can only be used in one direction—from the client to the server in that case—the former can send data straight to the server, but cannot receive it directly. For receiving data the client would need to tell the server, that it is ready to receive data. This message would have to be a an IPC call operation. When the server received data from its local channel, it can send this data to the client by using the reply IPC operation.

On the server side, the corresponding action for receiving data from the client would be to wait for an incoming message that contains the respective data. For sending data, the server needs to wait for an incoming call from the client, which signals that he is ready to receive something. The server can then send the data.

In order to distinguish both incoming messages in the server side, the client would need to send some sort of identifier first, describing whether he has attached data to the message (send case) or is waiting for data in the reply (receive case).

C2) *Use a bidirectional IPC connection.*

Another way of mapping an L4Go channel to L4 IPC would be to establish a truly bidirectional connection between the two tasks of interest. For that, two IPC gates must be present: the first is used for receiving on the client side, the second for receiving on the server side. Using this strategy, sending data is always possible from either task.

This approach is more symmetric and straightforward to implement than the first one,

[10] It should be noted that as Go does not have support for generics or template-like mechanisms, reflection is used here for receiving the data, i.e., `c` is not a channel object in this context, but some reflection data structure derived from the actual channel object that can be used for receiving the data using its `Recv()` method. See 2.1.5.1 on page 5 for further information on reflection.

because in order to achieve the same operation (send or receive) each side has to do essentially an identical sequence of actions.

When thinking about these two approaches more deeply, it became clear that the first one cannot work reliably under all circumstances. To understand why this is the case, consider the following code for two communicating tasks, a client and a server, that are connected by an L4Go channel, `c`:

```
for i:=0; i<42; i++ {
  // ...
  c <-i
}
```

Listing 3.1: Client, Go routine 1

```
for {
  i := <-c
  // ...
}
```

Listing 3.2: Client, Go routine 2

```
for {
  i := <-c
  c <-i+1
}
```

Listing 3.3: Server

The client contains two Go routines: Go routine 1 for sending data (listing 3.1) and Go routine 2 for receiving data (listing 3.2). The server has only one path of execution, in which he does both, receiving and sending (listing 3.3).

In this example, the client sends increasing numbers to the server, which in turn receives them, increases them by one, and sends them back. In Go routine 2, the client receives the increased numbers and may process them further. This is a valid usage of Go channels in conjunction with Go routines. Setups where this kind of work-splitting on the client side into two Go routines for sending and receiving might occur include scenarios where the duration for handling client requests by the server varies extremely. In such cases receiving the results in the same order the requests were sent is often not pratical. By having this partitioning on the client, these special cases can be accounted for.

When executing this example using approach C1, the program will most likely deadlock. This deadlock is caused by the fact that the client performs—concurrently—a send *and* a receive operation to the server (because of the usage of two Go routines). The server, however, only has one IPC gate for serving requests. As requests to one IPC gate can only be served by one thread, because only one thread can be bound to it at a time, the behavior of the example given above depends on the order of execution of the two Go routines. If Go routine 1 starts to run first, it will send data to the server, which can receive it and send it back. If, on the other hand, Go routine 2 is executed first, it will issue a request for receiving data. The server will block on this request and will never be able to serve it, because in order to send something back it first would need to receive something. This is a typical race condition between the two Go routines.

Approach C2 does not have this problem. Due to the two IPC gates used for the implementation of an L4Go channel, it is possible to send and receive data concurrently. This way the server can still receive data, although the client already sent a request for receiving data itself—there is no chance for a deadlock.

For the reason just explained, bidirectional IPC connections using two IPC gates (approach C2) will be used for the implementation. Henceforth, I will refer to such constructs as *IPC channels*.

3.2.5.2 Multiplexing of Channels

When using IPC channels as the underlying communication primitive for L4Go channels, two possibilities arise, on how they interact:

M1) *Use one IPC channel for transfering data of multiple L4Go channels.*

By attaching meta data to the actual payload of a channel, it is possible to distinguish between the data of multiple channels. By distinction of the various data items, they can be multiplexed on one IPC channel and be separated, i.e., demultiplexed, again later. To manage this transfer through one channel, some sort of protocol would have to be implemented in order to forward the elements on the receiver side to the correct L4Go channel. One could attach a simple label—e.g., an increasing number representing each L4Go channel—to the actual data. Figure 3.5 illustrates the process of multiplexing.

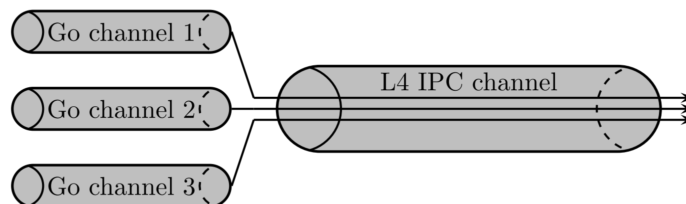

Figure 3.5: Multiplexing multiple Go Channels on one L4 IPC channel

M2) *Use an individual IPC channel for every L4Go channel.*

In this approach, every L4Go channel has exactly one IPC channel associated with it, which will be used for transfering channel items. This strategy relies on the fact that IPC gates can be created and exchanged dynamically at runtime. For each newly created L4Go channel, a new IPC channel needs to be established. An illustration can be seen in figure 3.6.

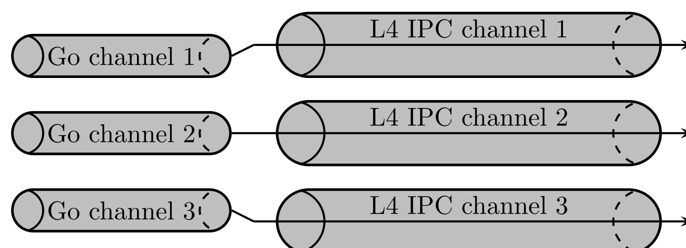

Figure 3.6: Each Go Channel has its corresponding L4 IPC channel

Making a decision between these two approaches is difficult. The first one, M1, has the advantage of using only one IPC channel for—at least theoretically—transfering data of an arbitrary amount of L4Go channels. This IPC channel could be statically allocated and so this

approach would potentially work even in scenarios where dynamic allocation of IPC gates is not possible.

But there are also disadvantages to this approach. The first being the more complex implementation, due to the additional bookkeeping involved for marking the payload with an identifier describing the Go channel it belongs to. This bookkeeping, as well as the multiplexing and demultiplexing associated with it, also induce extra runtime overhead. Second, deadlocks as explained before can occur, because if there is only one IPC gate, this can easily be blocked waiting for an operation to finish.

Approach M2, on the other hand, has no performance penalty caused by additional meta data processing and managing, but there might be increased memory usage due to the extra IPC gates involved. Also, it relies upon dynamic allocation of IPC channels in the case that L4Go channels are dynamically created. For a fixed amount of L4Go channels, however, a static setup is also possible: the IPC channel could be outlined at configuration time and then be used as the base of the L4Go channel.

A static setup is possible in both approaches and hence does not serve as a key distinction feature. Mostly due to the absence of possible deadlocks the decision was made in favor of approach M2.

3.2.5.3 Shared Memory Channels

The implementation as described before uses the UTCB for transporting the data to be sent over the L4Go channel. This has the advantage of being easily implementable using the previously discussed L4 IPC gates, because the Go IPC gate objects directly provide the means for writing data into the UTCB and reading from it (see sections 3.1.2.5 and 3.1.2.6, respectively).

However, the UTCB is of limited size: it consists of typically 63 message registers, i.e., machine words—depending on the machine architecture. This can prohibit the usage of L4Go channels for transfering large Go objects, like dynamically allocated arrays, lists or trees—as there might not be enough space for storing the whole object at once and transfering it.

One solution for this issue would be to transfer the object in multiple steps, i.e., sending the first part by filling the entire UTCB with data, then continuing with the next part until the entire object is transfered. The receiver would need to make sure that it correctly reassembles the object after receiving all parts. Although feasible, this implementation would significantly increase the complexity of the send and receive code.

Another approach is to use an area of shared memory for the actual data exchange. This memory would need to be shared between the tasks that are connected by the L4Go channel. The sender would write data into this memory and the receiver could read the data. The IPC gates would be used for synchronizing access to the shared memory, such that no reads at the receiver side conflict with writes from the sender. The shared memory area could theoretically be unlimited in size[11] allowing Go objects of arbitrary size to be sent.

[11] Sharing is possible only at the granularity of a page, so the minimum size of a block of shared memory would be one page, which is typically 4096 KiB—depending on the architecture. The maximum size is limited only by the amount of physical memory that is available to the system or, in the case of a system with swapping, by the amount of free secondary storage or the size of the virtual address space, respectively—whatever is smaller.

3.2.6 Runtime Library Modifications

Unfortunately, it turned out that a naive implementation in the proxy-style fashion as described in section 3.2.4.3 has two severe problems. Both are caused by missing synchronization between Go user code and the proxy channel implementation and require a deeper consideration. These two problems are:

P1) *A race condition that can lead to wrong data items being read from an L4Go channel.*

Figure 3.7 illustrates this problem. It shows two tasks—a client and a server. Only the client is of concern here, but the implementation on the server side is no different and, thus, equally affected by this issue. It contains two parts, a Go part which is written by the user, as shown on the left side, and a part that is used for implementing the L4Go channel and is working in the background, shown on the right. The first part is "ordinary" serial code, which contains a channel, c, that is connected to the server. Through this channel, a value is sent. After some additional work, a receive will take place. The second part, on the right, consists of two Go routines executing an endless loop: the first is used for sending data to the remote server task and the second is used for receiving data from it. As only the send part is relevant here, the receive part is left out and will not be considered any further. What the send part does, is to receive an item from the local channel c and send it to the server task (see section 3.2.4.3 for a more detailed explanation of the whole process).

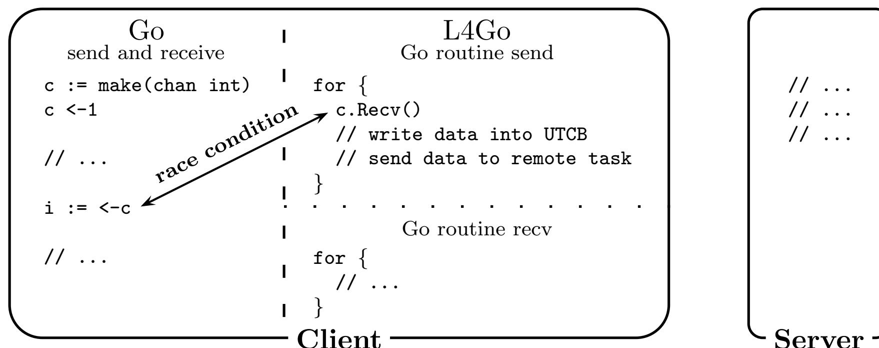

Figure 3.7: An L4Go channel implementation that may suffer from a race condition

The thoughtful reader may have noticed the occurrence of two "receives"—one in the user code and one in the send Go routine. Although these two receives look different, the left one using the arrow syntax and the right one using an ordinary method call, they both achieve the same goal—reading a value from the channel—by use of the same underlying functionality. These two receives block until an actual data item is written into the channel and can be read from it, and can occur entirely concurrently. If this happens, it depends on the scheduler's decision, which Go routine to wake up first—the one executing the user code or the one waiting for receiving data in the background. As only one of the Go routine's receive requests can be served, there exists a race condition among them about who the "winner" will be.

In order to cope with this problem, it is necessary to provide an additional synchronization point that tells the L4Go channel implementation, when the user actually wants to receive data, e.g., by usage of the arrow syntax for receiving data. Only if this happens, the receive part on the right side, i.e., the `c.Recv()`, would actually be executed.
Sadly, changes to the runtime library of Go are necessary, in order to implement this behavior, because otherwise there is no way for the implementation to tell whether the user is ready for receiving a channel item. Without this knowledge, the afore-mentioned call would have to block upon the receive operation which would inevitably lead to the race condition just described.

P2) *A synchronization problem that invalidates the semantics of L4Go channels.*

As explained before, Go channels can be used for synchronization between Go routines (see section 3.2.1 on page 25). This fact is captured by requirement R3. In a straightforward implementation of L4Go channels employing the proxy channel approach, however, this requirement cannot be satisfied.[12] It is violated because a send operation on such a proxy channel can unblock too early—before the element was actually received on the receiver side.
To understand why this can happen, have another look at figure 3.4 on page 31. The first step consists of sending an element, the integral number one in this case, using the arrow syntax (1). After that, it will be received transparently in the background (2). This `c.Recv()` method call will read the item from the channel and assign it to a variable. At this very moment of receipt, the send operation (1), which was issued by a different Go routine, will unblock, because the element was removed from the channel—it was received by some receiver, but in this case, it has not reached its final destination in the server task, but is only at a necessary intermediate location. This behavior clearly conflicts with requirement R3 from the user's point of view, because the item is not yet received by the server task.

To solve this synchronization problem, the send operation would need to block until the item was actually received in the server task. This could be achieved by adding a call to a callback function to the implementation of the send operation, which would have to block until the receiving task actually acknowledged the receipt, i.e., replied to the call. Again, as the send functionality is part of the runtime library, the latter would have to be adapted to fix this issue.

3.2.6.1 Go Channel Implementation

In order to cope with the two issues mentioned above—the race condition P1 and the synchronization problem P2—adaptations of the libgo runtime library were inevitable. In order to understand the actual implementation of the provided solutions, it is necessary to understand the basic mechanisms of the libgo implementation with respect to the sending and receiving of data using Go channels.

[12] In fact, netchan—using the same approach—suffers from this problem as well. It might, however, not be much of a concern there, because netchan's remote channels are more targeted at communication between systems and not within one and the same system but between tasks, as in my case.

The libgo runtime library provides the functionality for sending and receiving data from channels. The implementation consists of several functions for blocking and non-blocking send operations, as well as the blocking and non-blocking receive counterparts. There are also optimized versions available for channels that transport objects being 8 Bytes or less in size. When doing a send or receive in Go code (using the arrow syntax), the compiler directly emits the corresponding call to the libgo function.

Before an actual send action takes place, all libgo send functions call a *send acquire* function for requesting exclusive access to the channel and notifying other receivers that a send is being performed. After executing the send, a *send release* function is called, signaling possible receivers to wake up and relinquishing the exclusive access to the channel.

The receive action works in a comparable way, but the semantics of the operations differ. The *receive acquire* function also requests exclusive access to the channel. It then inserts the channel into the list of receivers waiting for a send to happen. After the actual receive is performed, the corresponding *receive release* function notifies any synchronous sender about the success and releases the lock that was used for granting exclusive access, just as in the send case.

3.2.6.2 Hooks

Having understood that part of the Go channel implementation, it is now possible to find a solution to the problems described in section 3.2.6. My solution comprises three hooks that were added to the libgo runtime library. Every hook consists of a semaphore and a corresponding callback function. The callback function will be set to a function operating on the semaphore, i.e., increasing or decreasing its value and thereby blocking itself or unblocking other callers, if the channel is an L4Go channel. For "normal" Go channels (which do not have any of the problems) it will be nil and not executed. Distinguishing between these two channel types is possible using the initialization function used for "connecting" a channel to another L4 task—if it is called, the channel is an L4Go channel, otherwise it is a normal Go channel.

Every hook is split in two parts, an internal one that is called by the runtime library, i.e., in the corresponding send or receive function, and an external one which is called by the L4Go channel implementation. Furthermore, one of these parts decreases the semaphore, i.e., waits on it, and the other one increases it, i.e., posts on it and wakes up other waiters. Which action is associated with which part depends on the semantics of the hook. The following three hooks are provided:

Pre-Send-Hook: The Pre-Send-Hook is used to signal the channel implementation that there is actually a send operation waiting that was issued by the user. Only in this case, the internal Go routine will receive a value from the channel in order to send it to the remote task. For this to work, the L4Go channel blocks on the associated semaphore until the runtime unblocks it within the send acquire function. In conjunction with the Pre-Recv-Hook, this hook solves the race condition problem P1.

Pre-Recv-Hook: The Pre-Recv-Hook signals the channel implementation that the user issued a receive operation. By waiting on the associated semaphore before the actual receive part takes place, the L4Go channel implementation is able to postpone the corresponding IPC operation for receiving data. Together with the Pre-Send-Hook, this solves the race condition problem P1.

Hook	Pre-Send-Hook	Pre-Recv-Hook	Post-Send-Hook
Internal part Executed in...	post send acquire	post receive acquire	wait send release
External part Executed ...	wait before sending	wait before receiving	post after sending

Table 3.2: Three hooks added to the libgo runtime library

Post-Send-Hook: The Post-Send-Hook is used to signal that the sending of data is complete. After the runtime library finished its process of sending data, it will wait on the semaphore associated with this hook. The L4Go channel will post on this semaphore when the actual send operation to the destination task is finished, i.e., the call got a reply. Using this construct, the synchronization problem P2 is resolved.

Table 3.2 shows a summary of the different implementation parts of the three hooks mentioned above.

3.2.6.3 No Hook Versions

The modified send and receive functions in the libgo library affect every send and receive action taking place, i.e., not only the compiler emitted calls but also reflective ones. This is the case because they are all implemented using the same basic functionality as provided by libgo, i.e., the previously mentioned various send and receive operations with their send acquire, receive acquire, send release, and receive release parts. The reflective methods are used for instance in the L4Go channel implementation and are provide by the reflect Go package (section 2.1.5.1 gives a brief introduction to this package). They can be seen in figure 3.4, where they appear at steps (2) and (5) (`c.Recv()` and `c.Send(data)`, respectively).

Having these reflection functions always execute the previously described hooks will not work, because this confuses the L4Go channel implementation by additionally waiting or posting on the semaphores in cases where no synchronization is actually necessary.

To solve this problem, two additional versions of the `Recv()` and `Send()` methods are introduced that do not execute the hooks just explained: `RecvNoHook()` and `SendNoHook()`, respectively. In order to provide these new methods, I made small adaptations to the netchan package. The L4Go channel implementation was then adjusted to use these new methods.

3.3 Keyboard Driver

To show that my work is suitable for developing services and drivers for L4, I decided to implement a keyboard driver. Special focus is laid on the usage of L4Go channels for the implementation, to demonstrate their usage. This section describes the driver that was developed.

The whole keyboard driver consists of two parts: the actual driver, which waits for keyboard interrupts to decode the actual data describing the keyboard event and to forward this event to a client. The client—the second part—is the component that waits for such events from the driver and performs a certain action. The following sections explain both components in more detail.

3.3.1 Driver

The driver's job is the receipt of interrupts and the preprocessing of event-related data, for instance, the scancode, for providing a general and keyboard independent event or data format. It will allow clients to subscribe for receiving these transformed keyboard events in order to provide keyboard related functionality themselves.

Keyboards interact with the system they are connected to by triggering interrupts. These interrupts are generally received by the operating system. An interrupt signals that data describing the keyboard event is available. This data is called a scancode. Scancodes describe the key that belongs to the corresponding event, the state, i.e., whether the key was pressed or released, and may contain additional data as well. A driver is notified when an interrupt occures and will read and interpret the scancode.

In the past several keyboard types were introduced that employed different sets of available keys and used different data format schemes: in 1981 the *IBM PC/XT* Keyboard was put on the market, in 1984 came the *IBM AT* Keyboard, and in 1987 the *IBM PS/2* Keyboard [Cha03b]. Each supports a different scancode set, the sets 1, 2, and 3, respectively, that are not compatible among each other. However, as the IBM PS/2 keyboard supports format 3 optionally, but also offers the possibility of choosing format 2, the AT and PS/2 keyboard can be regarded as compatible. The keyboards this driver aims to support are all IBM AT keyboard types. This means that scancode set 2 is used and must be supported (an overview is provided by Chapweske [Cha03a]).

The first step in the design of the driver is to decide on an event format which is abstract enough to be not specific to one particular type of keyboard but still is able to capture the important details of the ones of interest. The format chosen here assigns a keycode to every possible key: a symbolic constant that is different for every supported key—this way a possible client can handle keys in a comfortable way, e.g., within switch statements by enumerating the various keycodes and specifying the action to execute. In addition to that, a flag describes the state of the key, i.e., whether it was pressed or released. Both values are combined in the `Keyevent` Go type.

The implementation of the actual driver on L4 is straightforward and does not provide much room for variations: one has to request the keyboard's hardware IRQ from the operating system, in order to be able to receive key event notifications, as well as the associated I/O port for reading the keyboard scancode. Notifications for this IRQ can then be received using an object of the RecvIrq type as described in section 3.1.3.2 on page 24. When an IRQ is received, a scancode is available on the I/O port. Reading data from the I/O port is achieved using a Go wrapper of the

corresponding L4 function for reading a byte from a given port. After the data, i.e., the scancode, is read, it is decoded and transformed into a keycode and press-flag as described above. This process is accomplished using a lookup table like structure. In a last step, a `Keyevent` object will be created from the decoded data and be sent to the client.

3.3.2 Client

The client in this sample keyboard driver performs a very simple job: it will subscribe to the driver's server part to get notified about key events and will print the corresponding key code and press-state of the key of interest to the screen. For receiving key events, a Go channel is employed: it connects the client to the server and is used to transport objects of type `Keyevent`. In order to keep a good structure and isolation of components, the client is implemented as a separate task. For the communication to work under these circumstances, the channel connecting the client and the actual driver needs to be an L4Go channel, i.e., a Go channel that can be used for transfering data between tasks (see section 3.2). Although not mentioned before, the server part uses an L4Go channel as well to which `Keyevent` objects will be written.

In order for the client to get notified about any key events happening, it will issue a receive operation on the channel. If there is no key event available yet, the client will block, i.e., the execution of the Go routine trying to receive will be stopped and another Go routine scheduled. It will wake up whenever a keyboard IRQ was received, handled by the driver, and a corresponding `Keyevent` object sent through the channel. Listing 3.4 shows a stripped-down but fully functional version of the client as implemented for this work.

```
1  package main
2
3  import "fmt"
4  import l4c "l4go_chan"
5  import l4k "l4go_keyboard"
6
7  func main() {
8    channel := make(l4k.Keychannel)
9    l4c.Connect(channel, "keyboard_server", true)
10
11   for {
12     e := <-channel
13
14     fmt.Println(e)
15   }
16 }
```

Listing 3.4: A simple Go keyboard client

3.4 Problems

During the implementation of L4Go channels, several problems occured. Two of them are particularly interesting and will be described next.

3.4.1 IPC Cancelation

Go uses a Garbage Collector (GC) for memory management. As explained in section 2.3.2 on page 10, the GC had to be adataped in order to use L4 mechanisms instead of POSIX signals for interrupting Go routines that are to be garbage collected.

The adjusted implementation uses a special system call, `l4_thread_ex_regs_ret()`, to modify the state of the thread the Go routine is currently running on in order to make it execute code to mark all used Go objects, such that unmarked objects can be cleaned up later.[13] A special property of this system call is the fact that it will cancel any in-progress IPC operations. This cancelation has far-reaching consequences:

1) *Any canceled IPC needs to be detected and transparently restarted.*

   The interruption of a Go routine by the GC happens completely transparent to the program. If it noticed this interruption, some state would have been modified and its behavior could depend on the timing of these interruptions. Go programs that utilize, for instance, the previously described kernel objects (see section 3.1), use IPC messages to pass commands to the command thread that is used for implementing them. These IPC actions need to be restarted in a way the application does not notice, if the Go routine that issued the request to the command thread, is interrupted. On L4, the detection of a canceled IPC operation is possible by evaluation of the return value of the corresponding system call. A special value of `L4_IPC_SECANCELED` or `L4_IPC_RECANCELED` signals the cancelation of a send or receive IPC action, respectively. By checking this return value on both sides of the operation, i.e., on the side of the sender and on the side of the receiver, it is possible to simply re-execute the system call for the IPC operation using the same arguments as before.

2) *Restarting IPC operations only works reliably in the non-nested case.*

   On L4, IPC call operations can be nested. Figure 3.8 illustrates the concept of these nested operations. Object 1 is the initial caller. It does an IPC operation to Object 2. This in turn may, after doing some work, start a call operation to yet another object, Object 3.
   In order for the call initiated by Object 1 to return, it must be replied to. This can only be done by Object 2. For this to happen, however, Object 3 must first reply to Object 2, in order to unblock it from its call invocation.

   This concept of nesting calls may not work in conjunction with the restart mechanism as described in the first point. There are two cases to consider here:

   - *The call from Object 1 to Object 2 might get canceled, e.g., due to an invocation of `l4_thread_ex_regs_ret()` executed from the outside on the calling thread,*

[13] This procedure is commonly referred to as the mark-and-sweep garbage collection algorithm [Jon96].

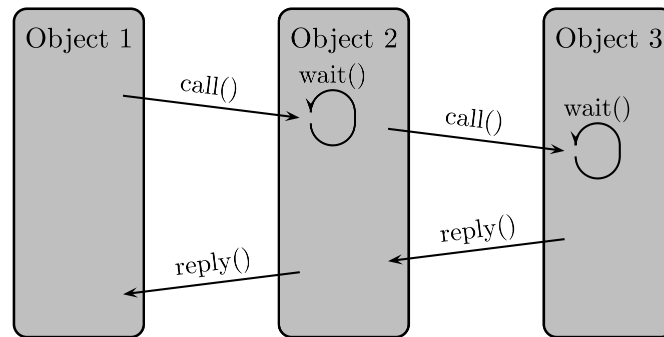

Figure 3.8: Nesting of multiple IPC call operations

before Object 2 was actually ready for the incoming IPC, i.e., it did not issue an IPC receive operation and both partners did not yet rendezvous.

In that case the canceled call can just be restarted as described in point 1. There is no need for any special handling on the receiver side, because the cancelation took place before Object 2 actually issued the corresponding receive system call.

- *Object 2 already received the incoming IPC and is blocked by its subsequent call to Object 3.*

  If this is the case, the call coming from Object 1 cannot simply be restarted, because there is no longer a corresponding receive operation at Object 2 for it is already finished and a rendezvous happened. Now, due to the cancelation of the incoming call, the reply capability which was created for it is no longer valid. Due to that, the reply from Object 2 to Object 1 can never be successfully executed.
  In order to solve this issue, the reply operation that would be issued by Object 2 is replaced with a "normal" IPC send operation. For this to happen, the receiver of the send, i.e., the issuer of the call (Object 1 in this case), must be known to Object 2. This is easily achievable by having the caller put its capability into a special memory area that is known to the receiver, right before the actual call operation takes place. The receiver can then always reply by using a send operation with this capability, instead of doing a true IPC reply.

These consequences described above influence the implementation of the Go kernel object wrappers as depicted in section 3.1. These wrappers represent the places where IPC operations are performed, that might get canceled due to interruption by the GC, as IPC there is initiated between Go code, i.e., from within a Go routine, and native C code. To tackle that problem, every IPC operation crossing that boundary had to be wrapped to incorporate the solutions as stated above.

After adaptation, objects of these types can safely be used from any Go code, even in the presence of a GC possibly interrupting Go routines any time.

3.4.2 Initial Memory Allocation

In order to serve future memory requests for Go objects faster, libgo's memory allocator pre-allocates a block of memory from the operating system, from which it can hand out pieces of memory to clients.

In the 32 Bit version of libgo, this initially allocated block has a size of ≈800 MiB.[14] This leads to problems in conjunction with L4Re, where the allocation of a consecutive blocks is limited to a size of 256 MiB. Due to that restriction, the part of L4Re responsible for these allocations, the l4re_vfs, had to be patched in order to allow for larger memory requests.

Another approach would be to adjust libgo's memory allocator in order to request a smaller chunk of memory initially or employ a completely different memory allocation strategy. But again, as changes to the Go runtime library should be avoided or kept as few as possible, this approach was withdrawn.

In addition to that, this amount of allocated memory is a per-task value. This means that each started task will request this amount of memory again. As main memory is a limited resource, this can lead to further allocation errors if there is not enough physical memory left to the operating system that can be handed out to user tasks. This will also directly limit the amount of Go programs that can be started in parallel, because startup will fail if this block of memory cannot be allocated.

This is not a critical problem in current setups which were regarded here, as enough main memory exists for all applications to startup correctly. However, if larger numbers of Go tasks need to execute in parallel, adaptations of the allocation scheme of libgo would be inevitable in order to not run out of memory.

[14] The 64 Bit version uses a different memory allocation scheme and, thus, does not have this issue.

4 Evaluation

In this chapter I present experiments in order to evaluate my work. In section 4.1 I analyze the overhead of IPC using the implemented Go objects over native L4 IPC using functions directly provided by L4Re. I also compare the performance of "normal" Go channels to L4Go channels. The subsequent part, section 4.2, discusses whether the Go programming language is suitable for systems development and compares it in terms of code size to languages traditionally used.

4.1 Performance

As a major part of my thesis includes the design and implementation of L4Go channels, they have to be analyzed in terms of performance. These channels use the L4 IPC functionality for transfering data across task boundaries, which is wrapped in Go objects for comfortable and easy usage (see section 3.1.2). Hence, the first step of the performance evaluation is a comparison between these Go objects and native IPC calls using the L4 C API. The second part will focus on the channels themselves and compares their performance to "normal" Go channels.

All measurements were performed on a machine with an Intel® Core™ 2 Duo E6750 CPU with 2.66 GHz and 2 GiB of main memory. The kernel and userland used is L4/Fiasco.OC revision 40788 from SVN repository and the version of libgo was revision 179017 from SVN as well. All measured Go programs have the GC disabled.

4.1.1 IPC Performance

In order to compare the performance of the IPC Go objects, i.e., SendIpcGate and RecvIpcGate, to native L4 IPC, I used ping-pong benchmarks. Ping-pong refers to a message that is sent by a sender to a receiver, which will immediately send it back. I measured the number of clock cycles until this message arrives back at the sender.

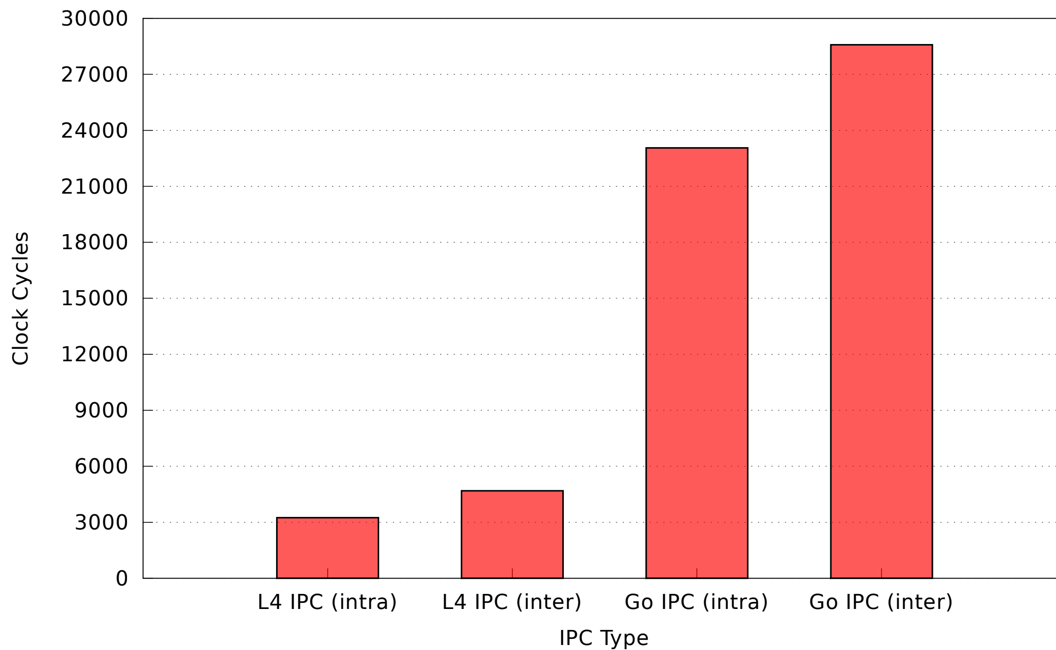

Figure 4.1: Native L4 IPC and Go IPC ping-pong performance

Figure 4.1 shows the measured performance of a ping-pong message—an IPC call on the one side and an open wait as well as a reply on the other—for four cases: native L4 IPC between threads within the same process and between different processes, and IPC using the developed Go objects for sending and receiving data, SendIpcGate and RecvIpcGate, again for the intra and inter-process case. All measurements were performed without any actual payload, i.e., no message registers were copied from the source to the destination. Shown is the average of 10000 repetitions, with implausibly high values being filtered out beforehand. This filtering was done automatically, removing all measurements that were twice as high as the minimum value. These high values are assumed to be caused by entering system mangement mode on the x86 architecture or by interrupt handling performed by the kernel in between. The distinction between intra and inter-process messaging is made, because the former is a special case of the latter that can be optimized specifically. Although intra-process communication is quite common in L4 programs written in C or C++, e.g., for synchronization between threads within one address space, in Go the usage of channels would be the preferred way to go in that case. Still, there are scenarios imaginable where the previously mentioned Go objects would also be used for communication between threads within one program, for instance, for applications that are meant to be distributed across several L4 tasks later on but with as few changes as possible.

As can be seen, the performance of IPC using the Go objects is nearly an order of magnitude slower than with the native IPC C API (3247 versus 23054 clock cycles for the intra-process version and 4683 versus 28580 in the inter-process case). This performance degradation can be attributed mainly to the additional communication between threads that are employed in the Go objects as well as the costs for scheduling. For the send case the former comes down to an additional call to the command thread (see figure 3.2), which in turn then issues the actual call

to the client. The client does an open wait, which causes three communications to be triggered: one to the command thread, one to the forward thread, and the actual wait call for incoming data from the server (see figure 3.3). After the wait operation, the client executes the reply, which again starts three IPC operations. As most of these operations are calls, there is a lot of scheduling involved, because the calling threads must be blocked until the receiver executes the reply. Furthermore, additional overhead is introduced for the transition from Go code to C++ and back. This is caused by additional wrapping layers, parameter encoding and decoding, as well as error handling.

As briefly explained before, the difference in clock cycles between *L4 IPC (intra)* and *L4 IPC (inter)* is caused by special optimizations within the kernel for this scenario. The relatively large difference in performance between the intra and inter-process case for the Go version of ≈5000 clock cycles, however, cannot be explained only be these optimizations, as this case is not optimized specifically in Go and only the L4 IPC kernel tweaks still hold. My explanation for this is the better locality in the intra-process case as well as the lack of task switches. Task switches are costly operations, that not only cause direct costs, e.g., for flushing cashes (especially the Translation Lookaside Buffer (TLB)) and scheduling, but also indirect costs, because more cache-misses occur due to caches not longer containing the correct data, which then has to be refetched.

4.1.2 Channel Performance

Based on the Go objects that wrap the L4 IPC mechanisms, L4Go channels were implemented (see section 3.2 for a detailed description of these channels). Next, I compare their runtime performance to that of "normal" Go channels operating within one task.

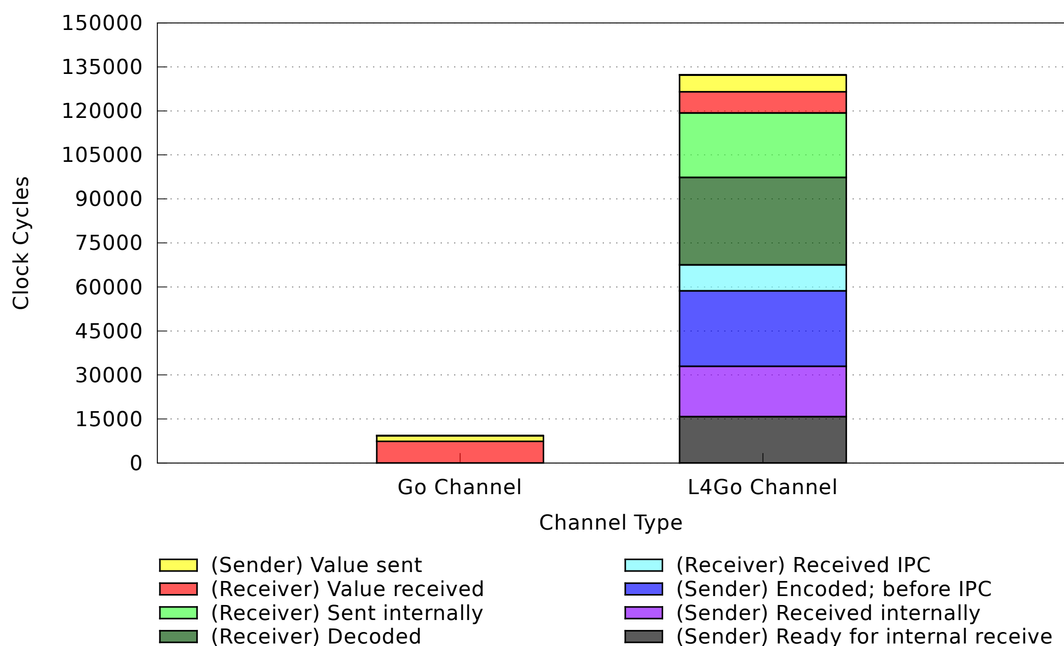

Figure 4.2: Go channel and L4Go channel performance

Figure 4.2 depicts the performance of both channel types, the Go channel that operates within one task and the L4Go channel that performs communication accross task boundaries (but can also be used for intra-process communcation). Shown are the average clock cycles of 10000 send operations of 64 Bit integral values from a sender to a receiver. The colored segments of the graphs illustrate the cycles consumed by significant steps in the process of transfer in the order they are executed.

The left bar graph shows the amount of cycles necessary for passing a small object between two Go routines. It distinguishes between the time it takes until the receiver is unblocked, i.e., has received the value (7384 cycles, shown in red), and the time until the sender is unblocked and continues to run subsequent code (additional 1944 cycles, shown in yellow).

The same action was performed using an L4Go channel that connects two tasks. It is shown on the right. As can be seen, the bar is split into more segments than the one on the left. These correspond to additional intermediate steps and are used to illustrate how the total amount of required cycles is made up for a complete send operation: The sender begins by issuing a request to send an object through a channel using the arrow syntax. After 15800 cycles (black), control flow is passed to the internally used Go routine and it was signaled that a value was written to this channel and can now be read. It takes additional 17144 cycles until the value is actually received internally from the channel (purple). Next, the value is encoded into a special format, such that type errors can be detected, and structural meta information are added (blue). This comsumes supplemental 25728 cycles and happens right before the value is written into the UTCB and the actual IPC call operation performed, that transfers the data to the receiver task. After 7552 clock cycles the receiver has received the incoming IPC (light blue). It decodes the value from its UTCB in 29840 cycles (green) and writes it to the channel in 21952 cycles (light green). After that, it takes 7224 until the receiver actually received the value in client Go code and can continue to run (red). The sender obtains control after another 5680 cycles (yellow).

As for the IPC comparison described in the previous section, the performance of inter-process channels is approximately an order of magnitude slower compared to standard Go channels. Some of the reasons for that are the same as well: a lot of blocking and scheduling is caused by several IPC call operations and other higher level synchronization primitives like semaphores and condition variables. As the previously evaluated Go wrapper objects are used for the actual data transfer, their overhead comes into play here as well. However, as shown in the figure, the actual IPC costs (light blue) are among the small contributors. Two major parts are the encoding and decoding of data using the gob package, making up a total of 55568 clock cycles ($\approx$42%). These two steps could be removed safely without loosing any properties of the channel during normal operation—they are only used to ensure type safety, i.e., detect transfer of incompatible objects between tasks. The other parts of the communication process are essential and would have to be profiled in more detail in order to find and fix bottlenecks.

In summary, it can be stated that although the performance of L4Go channels is not optimal, they are easily usable and fulfill their purpose: enhancing standard Go channels for means of inter-process communication. As was explained in section 3.2.4.2, performance was not a primary goal in the implementation, but rather a low amount of changes to existing code, and especially to the libgo runtime library, was desired.

4.2 Code Size

Another goal of the thesis is to evaluate the usability and practicability of Go for systems development in general and for L4 in particular. I will do this using a code complexity metric—the lines of code necessary to achieve a certain task.

The languages traditionally used for development of drivers and services on L4 are C and C++ and, if necessary, assembly for the associated architecture if certain features cannot be accessed by means of these other two languages directly. Most of the functionality provided by L4Re is implemented using these languages. In order to compare them to Go, I choose to compare two programs equal in functionality but implemented in C++ and Go, respectively, in terms of the Lines of Code (LOC) they are made up from.

LOC, sometimes also referred to as SLOC—Source Lines of Code—, is a metric to estimate the size of a software program by counting the number of lines of its corresponding source code. It is frequently used, for example, to estimate the number of man years necessary to write a certain software from scratch or to compare two software projects in terms of their "size" [AG83, CKHM12]. Although the LOC count is often considered a subpar metric for work estimation, due to its dependence on the formatting of the source code, and the programmer's coding style (e.g., the amount and placement of parentheses), as well as the numerous ways for solving a problem in general, most of the arguments against it do not hold in my case. For instance, the programs that are to be compared are all written by myself and in a consistent style. They also serve the same job by employing comparable means. As the LOC count is easily measurable, I choose to use it as the measure for comparison.

All LOC measurements were performed using the CLOC tool developed by Al Danial [Dan12]. It has support for a huge amount of programming languages, only some of which are actually used here. I will use the keyboard driver as explained in section 3.3 as an evaluation program. For that, I implemented a second version producing the same output using C++ as the development language.

Tables 4.1 and 4.2 show the LOC count as produced by CLOC for the keyboard drivers written in Go and C++, respectively. As can be seen, some additional files are listed there as well: the Go driver contains seven Makefiles and one Lua script. The Makefiles are needed as part of the integration into BID (see section 2.3.1). The Lua code is used as a start up script for running the compiled program on L4. The C++ version has comparable files for the same purposes. The Makefile and Lua values, as well as the numbers of blank and comment lines, are included for the sake of completeness here and are not of further interest.

The relevant parts for the comparison are the numbers for Go and C++ code (the latter being made up of the *C/C++ Header* and the *C++* table entries). According to these numbers, the Go version of the keyboard driver requires only two third of the lines of code of the corresponding C++ version (341 versus 510 lines of code for Go and C++, respectively).

This reduction in code size for the Go version in comparison to the one implemented using C++ can be attributed to two reasons:

- *Go code is slightly more compact than C++ code.*

  This fact is caused by syntactical differences between Go and C++. For instance, Go allows for multiple values to be returned *directly* from a function and assigned to variables, which is often useful for passing error codes or the like. In C++, one would need

Language	files	blank	comment	code
Go	5	51	111	341
make	7	35	0	39
Lua	1	6	1	16
Sum:	13	92	112	396

Table 4.1: Lines of code of keyboard driver written in Go

Language	files	blank	comment	code
C/C++ Header	3	40	43	258
C++	4	52	81	252
make	4	13	0	22
Lua	1	6	1	16
Sum:	12	111	125	548

Table 4.2: Lines of code of keyboard driver written in C++

to create a new data type for this purpose.
The defer-panic-recover mechanism [Ger10] used for resource deallocation in the case of an error is also less verbose when compared to the exception mechanism in conjunction with the Resource Acquisition Is Initialization (RAII) idiom as often used in C++ [Str00, Pib05].

- *The Go driver uses a high-level communication mechanism.*

  As explained in the corresponding section, the Go driver employs an L4Go channel in order to communicate keypress events. It abstracts from several details that have to be implemented explicitly in the C++ version—for instance, registering a new service in the environment and performing various sanity checks, e.g., regarding the protocol to use. There is also no need to worry about synchronization in the Go version, as the channel handles this transparently. In C++, one explicitly needs to perform a call on one side and a reply on the other.

This comparison shows, that it is indeed possible to write a compact driver for L4 using Go. All measurements have to be taken with a grain of salt, however, as it would also be possible to outsource low-level details of the C++ version into a library and provide a more abstract interface to this functionality (e.g., a similar mechanism to L4Go channels could be implemented), which would reduce the LOC count if this new library is assumed to be provided by the system and thus factored out from the evaluation.

5 Conclusion & Outlook

5.1 Future Work

During my work, many ideas for future work and improvements for my implementation came to mind, which could not be incorporated into this thesis due to timing constraints or because they were out of scope of my thesis. These include:

- *Investigating the possibility of migrating channels between tasks.*

  In native Go it is possible to create channels of channels, e.g., `chan chan int`—a channel that transports objects of type "channel of int". In the inter-process case, i.e., between address spaces, this migration is challenging. The difficulties include, in addition to the implementation itself which involves the mapping of IPC gates between tasks, making decisions about the ownership of a channel, i.e., who is responsible for a channel if a participating task is terminated, and the transfer of ownership.
  My implementation was designed with migration of channels in mind as there is a basic notion of a channel owner that is allowed to have a buffered channel—the client cannot have a buffered channel—but migration itself is not supported by now.

- *Wrapping of more kernel objects.*

  In this work, I focussed on the usage of L4 kernel objects that can be used for communication. But as explained in section 3.1, IPC gates and IRQs are not the only kernel objects available on L4. Further work could provide easy to use wrappers for these remaining kernel objects in Go.

- *Implementation of garbage collection for L4Go channels.*

  Go uses a Garbage Collector for releasing no longer used memory. As Go channels are language objects, they are automatically targeted by the GC as well. My extensions, i.e., the additional threads and IPC gates, however, are not cleaned up automatically because there is no standardized way to tell the GC what destruction method to call. Rather, the user has to release these resources explicitly.
  Future work could investigate the implementation of the GC and add support for automatic destruction of these L4Go channels. In addition, a standardized interface could be implemented that allows for registering destruction functions for any user-defined object, even accross different Go implementations.

- *Performance evaluation of full-fledged Go applications that use L4Go channels for various purposes.*

  The performance measurements of L4Go channels were performed in the form of micro-benchmarks, i.e., only the process of sending and receiving a small object in a tight loop

was considered. For better estimation of the impact of the performance of these channels on total application performance, measurements should be conducted for larger applications under real world conditions, which employ L4Go channels for achieving different goals: communication and synchronization (macro-benchmarks).

- *Further comparison of Go to C++ or other languages.*

  My evaluation provides a comparison between Go and C++ based on one metric that is measured on two functionally equivalent implementations of one application in both languages: the LOC count. For more reliable results, more programs of a wider variety of application areas should be compared. In addition, more metrics could be considered, for instance, the development speed and the average number bugs for a certain amount of lines of code.

- *Porting of Go to other systems.*

  For this thesis, the system to use—L4 with its L4Re—was predetermined. Future work could focus on porting Go to other operating systems. Ways for intergrating special features or mechanisms of these systems into the language could be found and the result compared in terms of effort necessary for the implementation and/or the performance achieved. This way, these system features of interest could be improved to be represented more easily in languages like Go.

- *Adjusting L4Re and the L4 kernel to ease the implementation.*

  Some of the problems that occured during the implementation were posed by the L4 system itself, for instance, the binding of certain kernel objects to threads that clashes with the way Go routines are managed. Further work could investigate changes to the system in order to avoid costly workarounds for these problems and to simplify the implementation of the kernel object wrappers and L4Go channels.

- *Implementation of L4Go channels within the libgo runtime library.*

  My implementation of L4Go channels is based on a proxy approach. The decision for this was founded mainly by the unnecessary high amount of changes that were required to the Go compiler and runtime library by alternative approachs. However, my solution showed that changes to the libgo runtime library are inevitable for preserving certain channel properties. Such being the case, L4Go channels could be implemented entirely within the libgo, as this could yield significant performance improvements, because of the removal of one level of indirection and the direct usage of the L4 IPC C API.

5.2 Conclusion

The goal of my thesis is to evaluate the suitability of the Go programming language for the development of services for the L4 microkernel, with focus on the integration of communication primitives provided by the language, and compare it to languages traditionally used in the area of systems programming.

For the implementation part, I did not have to start from scratch but could rely on my previous work that included the porting of the Go runtime to L4—allowing generic Go programs to run on the microkernel. My work comprises three main parts:

- *Wrapping of L4 kernel objects used for communication.*

  I represent IPC gate and IRQ kernel objects as objects in Go. This approach works well from a user's point of view, who wants to access these kernel objects using Go.
  The evaluation showed that the performance is nearly an order of magnitude lower compared to the IPC performance using the native C API. This performance degradation is mainly caused by communication between additional threads that were introduced. These threads were necessary due to the L4 specific characteristic of having the kernel objects bound to a specific thread—a feature that clashes with Go's Go routines, which can be mapped freely to operating system threads in a way the runtime library decides.

- *Adaptation of Go channels for incorporation of IPC on L4.*

  The extension of Go channels to send data not only between Go routines but accross processes boundaries is based on the Go kernel object wrappers. These L4Go channels preserve the special Go channel syntax and their synchronization semantics and hence can be used by developers in a familiar way to easily distribute work in Go applications accross several L4 tasks. Unfortunately, in order to preserve the Go channel semantics, it was inevitable to apply changes to the libgo runtime library—the gccgo compiler, however, could be left unmodified.
  The performance of these channels is also an order of magnitude slower compared to the standard intra-process Go channels. However, these channels were not implemented with high performance in mind and there are possibilities for optimizations. If high throughput is required, other means like communication via shared memory are available and should be preferred.

- *Implementation of a keyboard driver.*

  The developed keyboard driver showed, that it is possible two write drivers and services for L4 using Go. This driver was also used as the basis for a comparison with a functionally equivalent one written in C++. The evaluation showed that Go allows for a one third decrease in the lines of code compared to C++.

Glossary

ABI Application Binary Interface.

API Application Programming Interface.

BID Building Infrastructure for DROPS.

CLOC Count Lines of Code.

CML Concurrent ML.

DROPS Dresden Realtime Operating System.

Garbage Collector A program or routine for releasing no longer needed, i.e. referenced, memory. When a GC is in use, the user does not need to take care of releasing dynamically allocated memory.

GAS GNU Assembler.

GCC GNU Compiler Collection.

GNU Build System A set of tools for automating the process of building applications and libraries for UNIX based systems. It consists of GNU Autoconf, GNU Autoheader, GNU Automake, and GNU Libtool.

GPG GNU Privacy Guard.

IPC Inter Process Communication.

JNI Java Native Interface.

L4Env L4 Environment.

L4Re L4 Runtime Environment.

LD GNU Linker.

LOC Lines of Code.

MPL Meta Packet Language.

OCaml Objective CAML.

Plan9 A research operating sytem developed at Bell Labs beginning in the 1980s. The initial developers were, among others, Ken Thompson and Rob Pike who are also main initiators of the Go programming language [Lab].

RAII Resource Acquisition Is Initialization.

SLOC Source Lines of Code.

TCB Thread Control Block.

TCP Transmission Control Protocol.

TLB Translation Lookaside Buffer.

UTCB Userlevel Thread Control Block.

Bibliography

[AG83] A. J. Albrecht and J. E. Gaffney. *Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation*. IEEE Trans. Softw. Eng., 9(6):639–648, 11 1983.

[Ale01] Andrei Alexandrescu. *Modern C++ Design, Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2 2001.

[Cha03a] Adam Chapweske. *Keyboard Scan Codes: Set 2*. `http://www.computer-engineering.org/ps2keyboard/scancodes2.html`, 2003. [Online; accessed Tuesday 10th April, 2012].

[Cha03b] Adam Chapweske. *The PS/2 Keyboard Interface*. `http://www.computer-engineering.org/ps2keyboard`, 2003. [Online; accessed Tuesday 10th April, 2012].

[CKHM12] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. *Linux Kernel Development: How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*. `http://go.linuxfoundation.org/who-writes-linux-2012`, 2012. [Online; accessed Wednesday 28th March, 2012].

[Dan12] Al Danial. *CLOC – Count Lines of Code*. `http://cloc.sourceforge.net/`, 2012. [Online; accessed Sunday 20th May, 2012].

[fITS03] International Committee for Information Technology Standards. *International Standard ISO/IEC 14882 — Programming Languages - C++*. American National Standards Institute, 2nd edition, 10 2003.

[Ger10] Andrew Gerrand. *Defer, Panic, and Recover*. `http://blog.golang.org/2010/08/defer-panic-and-recover.html`, 8 2010. [Online; accessed Monday 21st May, 2012].

[Ger11] Andrew Gerrand. *C? Go? Cgo?* `http://blog.golang.org/2011/03/c-go-cgo.html`, 3 2011. [Online; accessed Wednesday 22nd February, 2012].

[Gro] Numerical Algorithms Group. *Calling C Library DLLs from C# – Utilizing legacy software*. `http://www.nag.co.uk/IndustryArticles/Calling_C_Library_DLLs_from_C_sharp.pdf`.

[Gro03] Operating Systems Research Group. *L4Env - An Environment for L4 Applications*. `http://os.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.pdf`, 6 2003.

[Gro10] Operating Systems Group. *L4 Runtime Environment*. http://os.inf.tu-dresden.de/L4Re, 2010. [Online; accessed Thursday 8th December, 2011].

[Inc09a] Google Inc. *Command cgo*. http://golang.org/cmd/cgo, 2009. [Online; accessed Wednesday 22nd February, 2012].

[Inc09b] Google Inc. *FAQ*. http://golang.org/doc/go_faq.html#overloading, 2009. [Online; accessed Tuesday 21st February, 2012].

[Inc09c] Google Inc. *Go For C++ Programmers*. http://golang.org/doc/go_for_cpp_programmers.html, 2009. [Online; accessed Saturday 21st January, 2012].

[Inc09d] Google Inc. *The Go Programming Language*. http://golang.org, 2009. [Online; accessed Monday 5th December, 2011].

[Inc09e] Google Inc. *Package fmt*. http://golang.org/pkg/fmt, 2009. [Online; accessed Tuesday 13th March, 2012].

[Inc09f] Google Inc. *Package gob*. http://golang.org/pkg/gob, 2009. [Online; accessed Wednesday 25th January, 2012].

[Inc09g] Google Inc. *Package io*. http://golang.org/pkg/io, 2009. [Online; accessed Tuesday 13th March, 2012].

[Inc09h] Google Inc. *Package netchan*. http://golang.org/pkg/netchan, 2009. [Online; accessed Monday 5th December, 2011].

[Inc09i] Google Inc. *Package opengpg*. http://golang.org/pkg/crypto/openpgp, 2009. [Online; accessed Tuesday 13th March, 2012].

[Inc09j] Google Inc. *Package reflect*. http://golang.org/pkg/reflect, 2009. [Online; accessed Monday 6th February, 2012].

[Inc09k] Google Inc. *Package zip*. http://golang.org/pkg/archive/zip, 2009. [Online; accessed Tuesday 13th March, 2012].

[Inc09l] Google Inc. *Package zlib*. http://golang.org/pkg/compress/zlib, 2009. [Online; accessed Tuesday 13th March, 2012].

[Ing02] Brian Ingerson. *Inline::C*. http://search.cpan.org/~ingy/Inline-0.44/C/C.pod, 2002. [Online; accessed Wednesday 22nd February, 2012].

[Jon96] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1st edition, 9 1996.

[Kau06] Bernhard Kauer. *L4Env-core Interface*. http://os.inf.tu-dresden.de/opentc/download/l4env-core.pdf, 5 2006.

[LA10] Jork Löser and Ronald Aigner. *Building Infrastructure for DROPS (BID) Specification*, 4 2010.

[Lab] Bell Labs. *Plan 9 from Bell Labs*. `http://plan9.bell-labs.com/plan9`. [Online; accessed Thursday 8th December, 2011].

[Li04] Peng Li. *Safe Systems Programming Languages*. 2004.

[Lia99] Sheng Liang. *The Java™ Native Interface – Programmer's Guide and Specification*. Addison-Wesley Longman, 7 1999.

[Lie96] Jochen Liedtke. *L4 Reference Manual - 486, Pentium, Pentium Pro*. 1996.

[MHD+07] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. *Melange: creating a "functional" internet*. SIGOPS Oper. Syst. Rev., 41(3):101–114, 3 2007.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[MYS03] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. *Capability Myths Demolished*. 2003.

[Pib05] Roland Pibinger. *RAII, Dynamic Objects, and Factories in C++*. `http://www.codeproject.com/Articles/10141/RAII-Dynamic-Objects-and-Factories-in-C`, 4 2005. [Online; accessed Monday 21st May, 2012].

[Poh08] Aaron Pohle. *Eine Shell für L4Env*. Belegarbeit, Technische Universität Dresden, 5 2008.

[Rei91] Martin Reiser. *The Oberon System. User's Guide and Programmer's Manual (ACM Press)*. Addison-Wesley, 5 1991.

[Rep91] John H. Reppy. *CML: A higher concurrent language*. SIGPLAN Not., 26:293–305, 5 1991.

[RRX09] John H. Reppy, Claudio V. Russo, and Yingqi Xiao. *Parallel Concurrent ML*. SIGPLAN Not., 44:257–268, 8 2009.

[RW92] Martin Reiser and Niklaus Wirth. *Programming in Oberon - Steps beyond Pascal and Modula*. Addison-Wesley, 3 1992.

[Sco09] Michael L. Scott. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009.

[Str00] Bjarne Stroustrup. *The C++ Programming Language: Special Edition; Appendix E: Standard-Library Exception Safety*. Addison-Wesley Longman, 3nd edition, 2 2000.

[Sut02] Herb Sutter. *The New C++: Smart(er) Pointers*. http://www.drdobbs.com/184403837, 8 2002. [Online; accessed Thursday 31st May, 2012].

[Tea] The GCC Team. *GCC, the GNU Compiler Collection*. http://gcc.gnu.org/. [Online; accessed Thursday 8th December, 2011].

[Tho] Ken Thompson. *Plan 9 C Compilers*. http://plan9.bell-labs.com/sys/doc/compiler.html. [Online; accessed Wednesday 4th January, 2012].

[WG92] Niklaus Wirth and Jörg Gutknecht. *Project Oberon. The Design of an Operating System and Compiler (ACM Press Books)*. Addison-Wesley, 11 1992.

[Wik11] Wikipedia. *Go (Programming Language)*. http://en.wikipedia.org/w/index.php?title=Go_(programming_language)&oldid=484405078, 2011. [Online; accessed Wednesday 28th March, 2012].

[Wir88] Niklaus Wirth. *The programming language Oberon*. Softw. Pract. Exper., 18(7):671–690, 7 1988.

[WL11] Alexander Warg and Adam Lackorzynski. *Rounding Pointers – Type Safe Capabilities with C++ Meta Programming*. Sixth Workshop on Programming Languages and Operating Systems, 10 2011.