Message Passing for Programming Languages and Operating Systems

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 141

Systems Group, Department of Computer Science, ETH Zurich

Message Passing for Programming Languages and Operating Systems

by

Martynas Pumputis

Supervised by

Prof. Dr. Timothy Roscoe, Dr. Antonios Kornilios Kourtis, Dr. David Cock

October 7, 2015

inf | Informatik
Computer Science

Abstract

Message passing as a mean of communication has been gaining popularity within domains of concurrent programming languages and operating systems.

In this thesis, we discuss how message passing languages can be applied in the context of operating systems which are heavily based on this form of communication. In particular, we port the Go programming language to the Barrelfish OS and integrate the Go communication channels with the messaging infrastructure of Barrelfish. We show that the outcome of the porting and the integration allows us to implement OS services that can take advantage of the easy-to-use concurrency model of Go.

Our evaluation based on LevelDB benchmarks shows comparable performance to the Linux port. Meanwhile, the overhead of the messaging integration causes the poor performance when compared to the native messaging of Barrelfish, but exposes an easier to use interface, as shown by the example code.

Acknowledgments

First of all, I would like to thank Timothy Roscoe, Antonios Kornilios Kourtis and David Cock for giving the opportunity to work on the Barrelfish OS project, their supervision, inspirational thoughts and critique.

Next, I would like to thank the Barrelfish team for the discussions and the help.

In addition, I would like to thank Sebastian Wicki for the conversations we had during the entire period of my Master's studies.

Finally, I would like to thank my girlfriend and my family for their support which allowed me to follow my passion.

Contents

Chapter 1

Introduction

Recently, message passing as a form of communication is getting more attention within domains of programming languages and operating systems.

Concurrent programming languages such as Go [14] and Erlang [6] make this form accessible as a first-class citizen and encourage synchronization among concurrent components by passing messages instead of sharing memory. It is a common believe that the latter approach to concurrency is more difficult to reason and as a consequence, is more error-prone [22]. In addition, the message passing systems can be formally described and verified by applying theories of process calculi [38] or the Actor model [34].

Meanwhile, increasing complexity, heterogeneity and ubiquity of multi-cores in the hardware landscape have led to research and development of operating systems which resemble more distributed systems than regular operating systems [17]. An example of such a tendency is the Barrelfish OS [16]. The OS is based on a multikernel approach which implies that each core is running a special isolated instance of a small kernel. One of the founding assumptions of the OS is that kernels do not share memory and instead, consistency and synchronization among kernels can only be achieved via channels-based message passing.

The goal of this thesis is to explore how (or whether) a message passing language can be incorporated in the context of a messaging passing operating system. In particular, how channels of the Go programming language can be integrated with Barrelfish's messaging infrastructure in order to provide the easy-to-use concurrency model of Go to the OS. In addition, how the integration can be employed for building Barrelfish OS services in Go.

We will start with Chapter 2 in which the relevant parts of Go and Barrelfish will be introduced. Following that, Chapter 3 will present our effort towards porting Go to the Barrelfish OS, as well as integrating the Go channels with Barrelfish's messaging infrastructure. In addition, the chapter will take a

look into implementing the OS services in Go. Next, in Chapter 4 we will evaluate the port by performing series of benchmarks. Finally, Chapter 5 will provide conclusions and directions for future work.

Chapter 2

Background

In this chapter we present background information on two topics used throughout the thesis: the Go programming language and the Barrelfish operating system. The former is introduced in Section 2.1 and an overview of the latter is given in Section 2.2.

2.1 Go Programming Language

Go (or *golang*) is a general-purpose open source programming language, with an emphasis on system programming. The language development initially started with Rob Pike, Ken Thompson and Robert Griesemer at Google in 2007 [13].

The main motivation for conceiving the language was to address the problems of a large-scale software development process at Google [45]. Firstly, it means to minimize build time by taking an alternative approach to dependency handling when compared to C/C++. Secondly, it aims to reduce a set of possible implementations for a single problem by providing a simple programming model. Thirdly, the language should enable writing tools targeting the language itself.

The simple specification and the programming model of Go is backed by one of the main principles behind the language - *do less, but enable more* [20]. The slogan is translated to a requirement to allow expressing solutions to a variety of problems despite the simplicity and the brevity of the language. Indeed, Go stands out by the size of its specification when compared to any other general-purpose programming language [14], e.g., Scala [48]. One can argue that the simplicity lead to increasing popularity and adoption of the language. For example, many emerging projects outside Google such as Docker [42] or CoreOS [19] picked Go as the main language.

Besides being a compiled, statically-typed and garbage-collected language,

Go brings to the table concurrency as a first-class citizen. The concurrency feature was the main influencing factor for choosing the language for this thesis project.

An example of a Go application is shown in Figure 2.1. The application prints the first 10 Fibonacci numbers. Generation of the numbers is done by the `fib` coroutine which runs concurrently with the `main` function. The communication between `main` and `fib` happens over the Go channel `c`.

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

Figure 2.1: The Fibonnaci numbers generator in Go.

The following sections give a brief overview of type system and concurrency primitives of Go. The overview should help the readers to better understand the upcoming parts of the thesis.

2.1.1 Types and Type System

Go supports basic types such as integers, floats, chars, bytes, pointers, strings, arrays and dynamic arrays called slices. Their representation in memory resembles the one of C with a few exceptions, e.g., strings and arrays are stored together with their length [23], which allows to perform runtime checks against overflow. Besides the primitive types, Go includes built-in hash tables and C style *structs* for type composition.

Instead of adopting class based inheritance, Go provides an alternative solution: *interfaces*. Interfaces are used to specify a list of required methods. A type satisfies an interface definition if it implements methods required by the interface. Go compiler is able to infer such information and does not require any additional information besides the interface declaration. In ad-

dition, every object implements an empty interface. The closest equivalent of the interfaces is *Type Classes* of Haskell [41] or *Traits* in Scala [48].

Figure 2.2 presents declarations of the `Stringer` interface and the `Animal` type which satisfies the interface by implementing the required `String` method.

```
type Stringer interface {
    String() string
}

type Animal struct {
    Name string
}

func (a Animal) String() string {
    return a.Name
}
```

Figure 2.2: An example of the interfaces in Go.

The Go type system is *strong* and *static*. A strong type system is one which does not allow one to apply operations to arguments of inappropriate types, while static is one in which all expressions, variables and methods have types and which can check type rules at compile time [43].

Go is able to infer data types. As a consequence, explicit type declarations are rare. In addition, due to the presence of interfaces, the type system can be considered as *structural* which means that type compatibility is determined by interface definition and not by name.

2.1.2 Concurrency

Go makes concurrent programming easier by providing two first-class citizens: *goroutines* and *channels*. The concurrency model that Go implements is influenced by ideas presented in *"Communicating Sequential Processes"* by C.A.R. Hoare [35].

Goroutines

Goroutines are concurrent lightweight processes which operate within the same address space as their parents. Spawning of a process in code is denoted by a `go` statement in front of a function call expression.

The Go scheduler is not *fully* cooperative, because a goroutine can be preempted when it does a function call. However, the goroutine hands off the execution to the scheduler in the case of *syscalls*, communication over blocking channels or it can manually yield to the scheduler. The preemption can

solve a resource starvation problem which happens when a long-running goroutine does not issue any operation listed above and as a consequence, other runnable goroutines cannot be scheduled.

Traditionally, goroutines are multiplexed over OS-level processes. The runtime (*RTS*) of Go can consists of more than one OS process. Therefore, some degree of parallelism can be achieved.

Each goroutine is associated with a stack frame which can dynamically grow and shrink during its lifetime. A mechanism to support that is called *split stack* [53]. The check for the grow happens in the beginning of a function call when the preemption might occur.

Channels

In Go, a mean for goroutines communication within the runtime is channels. In contrast to the *Actor Model* [34] which is implemented in Erlang et al., a channel has to be manually created in advance. Each channel is always typed and bi-directional. However, it is possible to add a directive to denote a channel as one-way (write-only or read-only).

A channel is either buffered or unbuffered. The main difference is that sending to an unbuffered channel will block until a receiver has received a message, while sending to a buffered channel blocks until a message has been stored in a buffer. A receive operation always blocks until there is some message to be received. To avoid blocking when receiving from empty channels, sending to full channels or channels from which nobody receives, Go introduces a *select* statement. The statement allows one to list multiple send and receive operation, one of which will proceed. The precedence is given to one which does not block. If all operations block, then the statement will block the goroutine. However, it is possible to use a default case which will be executed to avoid blocking the goroutine.

Because channels of Go are first-class values, they can be transferred over channels. Such a feature makes multiple interesting concurrency patterns possible [44]. For example, we can implement a worker pool by spawning multiple worker goroutines. Each goroutine is listening on the same channel. A message to this channel consists of a request and of a response channel. After handling the request, the worker sends a result via the response channel.

It is important to mention, channels do not escape the RTS boundaries. Therefore, they can not serve as a communication medium between separate run-time instances.

Finally, Go encourages to "*not communicate by sharing memory; instead, share memory by communicating*" [12]. However, the slogan looks a bit inconsistent,

because Go, in opposite to Erlang, provides a notion of shared variables and low-level locking primitives such as read/write mutexes or conditional variables to protect access to them.

2.2 Barrelfish Operating System

Barrelfish is a research operating system developed by ETH Zürich in collaboration with Microsoft Research [16]. The main principle of this OS is the *multikernel*: [17] an alternative approach to structuring an operating system. The OS is as a distributed system, such that each core of a machine is running a single instance of a small kernel, called a CPU driver. The kernels do not share any memory, even on cache-coherent machines. State consistency within the OS is achieved by using message passing. The kernel code does not necessarily have to be the same between cores. Hence, Barrelfish can run in heterogeneous environments.

The CPU driver resembles a microkernel, in that it provides a minimal trusted set of functionality including scheduling, resource protection and handling of platform-dependent events such as interrupts. The rest is implemented in user-space, along with device drivers, file system, handling of page-fault signals arriving from the kernel, etc., which in traditional monolithic operating systems is part of the kernel.

Each kernel runs the special user-space process, called the *monitor*. The process is responsible for coordinating message passing, supervising other processes and handling a state of the system. Together with the CPU driver, monitor forms a foundation of the system.

Barrelfish represents protectable objects such as physical memory, virtual memory address space, etc., using *capabilites*. The model is similar to the one provided by the microkernel *seL4* [59]. Each capability is typed and supports a set of operations.

The design of the OS is influenced by exokernels [28] [16], therefore a lot of functionality is exposed to user-level applications. As a consequence of that, Barrelfish provides a *libbarrelfish* run-time library implementing user-level threading, memory management, support for message passing, capability management and so on.

The model of the OS resembles the one of Go in a sense that both rely on the same communication form. However, Barrelfish's choice for the message passing is based on performance reasons and the fact that in the future we cannot assume the same premises for cache-coherency, memory hierarchies as we do now [17]. Meanwhile, Go prefers the message passing over the synchronization via shared memory for the safety reasons [12].

2.2.1 Dispatchers

A dispatcher is an entity which the CPU driver can schedule. In regular operating systems, the dispatcher would practically correspond to an OS process. However, Barrelfish implements a threading mechanism based on scheduler activations [27], also known as the "N:M" scheduling model. Consequently, threads are implemented in user space without kernel involvement. The default implementation is provided by *libbarrelfish*.

Upon scheduling a dispatcher, the kernel invokes an *upcall* to the user-level dispatcher which takes further scheduling decisions. The decision is based on the state of the dispatcher: *enabled* or *disabled*. In the enabled state, the dispatcher is running a user thread which is resumed after the upcall. In the disabled state, the dispatcher executes its management routines which can decide to run a user thread. One of the management routines is a page-fault or message arrival handler.

During their lifetimes, dispatchers do not migrate among cores. Therefore, we say that dispatchers are bound to the cores. In addition, dispatchers can be grouped into so-called *domains*. The advantage of having a concept of domain is that address space among dispatchers within the same domain can be shared.

Usually, dispatchers do not directly run user-level applications. Instead, they are responsible for scheduling preemptive user-level threads provided by libbarrelfish.

The user-level threads can be compared to goroutines in a sense that they both provide concurrency. However, the libbarrelfish's threads are more heavy-weight as they involve more hardware registers during the context switch. Also, a stackframe of the thread is static and does not change during its lifetime, which is not the case for a goroutine.

One way to run goroutines on Barrelfish is to schedule them directly on a dispatcher. However, the runtime of Go presumes the existence of threads and therefore such approach would require quite a few changes in the runtime. An additional obstacle to this approach is that the Go scheduler is not fully preemptive and thus, it would require further analysis of whether such scheduler could be run directly on the dispatcher.

The other way to run goroutines is to demultiplex them on the user-level threads. This approach does not require any changes in the runtime. However, it brings an additional level of scheduling which might have negative impact on performance.

2.2.2 Inter-Dispatcher Communication

The message passing in Barrelfish plays an essential role, not just because of its distributed architecture, but also the regular OS functionality is provided by services which run in a form of dispatchers. Therefore, an effective mean for communication between dispatchers running on the same or different cores is needed.

Barrelfish provides a unified interface for messaging based on channels. Each channel is bi-directional and typed. In addition, channels are built on top of an underlying *interconnect driver*. The two most common drivers are *Local Message Passing (LMP)* and *User-Level Message Passing (UMP)*.

The LMP driver is used only by channels of dispatchers running on the same core. The main idea behind LMP is that both sides of a channel are represented by endpoint capabilities. Sending to a channel translates to an invocation (system call) of a receiver endpoint capability. The following event is handled by CPU driver which, in consequence, does upcall to the receiving dispatcher.

The UMP driver is completely implemented in user-space and, opposite to LMP, it does not require any support from the CPU driver. The driver is used by dispatchers running on different, but cache-coherent cores.

The interconnect drivers do not expose a unified interface. However, Barrelfish provides a tool called *Flounder* which hides any implementation details of IDC from a user by generating messaging *stubs*. Each stub is defined by a special interface definition language. The language allows one to assign types to parameters of any message.

A channel binding is coordinated by the monitor process. Between parties, one acts as a *server* and the other as a *client*. Before any channel establishment, the server exports some service by announcing it to the monitor. The latter assigns a unique *interface reference (iref)* value. This value is used by the client when connecting to the server. After the completion of the binding, any message defined by the interface can be sent. In general, the low-level details of the establishment are handled by Flounder-generated stubs.

Receiving of messages is handled via the *waitset* mechanism. Each waitset can have multiple channels assigned to it. A receiving thread polls [1] a given waitset for upcoming message events and blocks itself if there are none. Such event-driven model inspired handling which is also know as "stack-ripping" [4] has advantages in terms of performance. However, we could argue that development by using such concurrency technique is more complex when compared to the Go concurrency model. Go allows us to

[1] Usually, the polling is done with the `event_dispatch` function.

handle blocking either by spawning a goroutine per blocking operation or handling such operations with the select statement.

Although Go and Barrelfish channels share many similarities, their scope of functioning differs: in Barrelfish they are used for inter-dispatcher communication, while in Go, they can be used only for intra-dispatcher communication, i.e. they cannot step over dispatcher boundaries. The further comparison of Go and Barrelfish channels can be found in Section 3.2.1.

2.3 Related Work

The rest of the chapter gives an overview of similar techniques which makes the messaging possible in the context of operating systems.

libthread The threading library of Plan9 `libthread` [1] provides concurrency primitives which include coroutines and channels. Both can be employed by OS processes. However, the underlying implementation of the channels implies shared address space. Therefore, in opposite to our approach, it does not allow processes which do not run in the same domain to communicate over channels. The concurrency primitives were influenced by the Alef programming language and they both directly influenced the design of Go [21].

Unix Pipes The traditional Unix pipes [39] provides similar communication tools to OS processes as the buffered Go channels to coroutines. A pipe makes a writer process block once the pipe's buffer gets filled. The same behaviour can be observed in Go. In addition, an attempt to send to a closed channel in Go results in panic, whilst in the case of pipe, the `EPIPE` is returned. However, in opposite to Go, the absence of the reader is notified to the writer by the `SIGPIPE` signal. In general, the communication over pipes requires some level of coordination by an operating system (e.g., allocating or controlling the access to a file descriptor), which is also the case for the channel establishment in Barrelfish.

netpoll The network handling via `netpoll` [8] in Go is similar to the channel integration proposed in the thesis: netpoll demultiplexes IO notifications to goroutines, while `EventDispatch` demultiplexes incoming Flounder messages. However, the former is integral part of the Go runtime, while the latter is implemented as a goroutine. In addition, netpoll is responsible not only for receive events, but also for send which is not a case for EventDispatch.

AC Harris et al. [33] propose a composable asynchronous IO for native languages. The extension called AC allows to handle asynchronous IO in a

different way when compared to the traditional event-driven model in Barrelfish. It introduces an `async` statement which performs IO operations in concurrent way without involving multiple callbacks. The `async` statement of AC can be compared to the `go` statement: both can be used to avoid blocking of a execution context. However, AC concentrates on the sequential programming model and it does not provide the same level of concurrency as our approach in Barrelfish does.

Clive The Clive operating system [15] tries to use the Go programming language for the OS development which is similar to our purpose. First, it extends the Go compiler and runtime in such way that channel termination can have a reason assigned to it which is accessible to participants. Next, it leverages the CSP ideas used in Go for structuring the operating system processes and for inter-process communication. The processes in the OS can communicate over channels in spite of their location. E.g. processes residing in different virtual address space or separated by network are able to communicate over channels. Finally, the project aims to make Go to run on bare-metal. Nevertheless, the latter is still work in progress. The main difference between channel based communication in Clive and Flounder over Go channels is that our approach requires channels to be typed, while the former assumes messages to be of the raw bytes type.

Chapter 3

Porting Golang to the Barrelfish OS

This chapter presents our approach of porting the Go programming language to the Barrelfish OS. First, in Section 3.1 we will introduce the effort of porting the Go compiler and the Go runtime. Next, Section 3.2 will present the integration of the Go channels with Flounder. Finally, in Section 3.3 we will give an overview of implementing the Barrelfish OS services in Go.

For this port, we assume that the domain of a Go program runs on a single core.

3.1 Initial Port

As it is mentioned in Section 2.1, Go is a compiled language. The typical compilation process produces a single statically linked binary which can be directly run by an OS. The executable binary consists of two parts: user code and run-time system. The run-time system is coupled with a Go compiler which, in addition, is capable of compiling the system. Therefore, when porting Go to a new operating system, compilation and run-time support steps must be taken into consideration first.

The bare minimal runtime of Go requires a small set of functionality from an OS. The set includes:

- Creating and yielding a schedulable context, e.g., thread.
- Delaying execution of a thread.
- Support for locking primitives in the form of semaphores.
- Memory allocation.

3.1.1 Choosing a Go Compiler

At the time of writing, there exist three distinct compilers which are compliant with the Go specification: LLVM based *llgo* [56], GCC based *gccgo* [54] and *gc* [10]. Therefore, as a first step of the port, a decision which compiler to use should be made.

First of all, we eliminated the llgo compiler from the list, because the project is experimental and its current state is not acceptable for the porting purpose [56].

Next, we considered the gccgo compiler. The compiler is a frontend to the GNU compiler collection. GCC is used to compile the Barrelfish kernel and user-space applications. Thus, the backend of gccgo is well tested on the OS. In addition, the compiler has decent, albeit limited interoperability with C/C++ [54]. Also, because of the backend, it supports a variety of processor architectures. Therefore, the compiler can be considered as a reasonable choice for the port. However, the most recent version of gccgo falls behind implementing the Go specification and, at the time of writing, the compiler supports only **1.2** version of Go. Furthermore, the compiler is not so widely used as gc.

Finally, the third option is the gc compiler. The compiler is a derivative of the Plan9 compiler family [10] and is used by the Go team to implement the most recent version of Go. Usually, it is referred as the *de facto* compiler within the Go community. Some parts of the run-time system which are shipped together with gc are written in an extended form of Plan9 assembly language. Therefore, for the sake of the complete compilation, the assembler is included into the Go distribution. Moreover, the gc compiler stands out by its simplicity. Nevertheless, gc follows a non-standard ABI convention. Thus, it is not directly interoperable with the SystemV ABI [57]. The decision to rely on a Plan9 tool was based on their simplicity [25]. Also, the authors of the language come from Bell Labs - the place where Plan9 was born.

The gccgo compiler is claimed to have better performance characteristics for CPU-bound applications [55]. Mainly, it is due to more sophisticated optimizations applied by the GCC backend as it is stated in [55]. However, the benchmarks performed in [18] refutes the claim.

In spite of the listed disadvantages of gc, we decided to use gc for the port because of the following reasons: support for the latest Go version, its popularity and its simplicity which becomes relevant in the context of the thesis.

To the date, there exist two major versions of Go: **1.4.2** (*stable*) and **1.5** (*beta*). The major difference between the two is that the newer version replaces C parts within the Go compiler suite and the run-time system by equivalents which are written in Go. As a consequence, it allows completely removing

the C compiler from the Go distribution. We chose to base our work on the beta version, because in the future it would be easier to merge any upstream changes of Go into the port. Also, the 1.4.2 version will not be backported in the future.

As a target platform architecture for the port, *x86_64* (*amd64*) is chosen, because Barrelfish supports it. Also, the machines we have chosen for the evaluation (see Chapter 4) are of this architecture.

3.1.2 Run-Time Integration

The run-time system of Go does a variety of heavy-lifting tasks including: scheduling goroutines, managing memory, providing support for channels, etc. An underlying implementation of the system is tightly coupled with the functionality exposed by the OS. Usually, in regular monolithic operating systems the functionality is provided to a user-space applications in a form of system calls (*syscalls*) and a common approach to access it is to use a standard library such as *GNU libc* [40].

In spite of this common practice, one of the first Go ports of the gc compiler to target the *Linux* operating system does not depend on libc. Instead, it accesses a system function by directly invoking a syscall to the kernel. Such an approach is possible for a monolithic kernel, because the kernel provides a lot of functionality to user-level applications and a stable syscall ABI. However, in the case of Barrelfish, functionality is distributed among the OS services and is accessed via the helper functions of libbarrelfish. Therefore, taking the same approach as in the Linux port, i.e. not relying on a library, would require duplicating quite a few parts of libbarrelfish. However, a duplication in software engineering is usually considered bad practice [37].

One way to avoid the duplication would be to use the older version of Go which implements some parts of the run-time system in C. Thus, the included Plan9 C compiler could be leveraged for compiling libbarrelfish. However, the compiler supports a subset of the C programming language [58] which does not cover all features used by libbarrelfish. Hence, this approach can not be taken.

The other option is to link the run-time against libbarrelfish. The recent port for the *Solaris* operating system takes a different path than the Linux port, by relying on libc. The library is dynamically linked against the RTS. However, for the Barrelfish port it is not possible to do it in the same way, because the OS does not yet support load-time relocation. Consequently, for our port, we decided to harness static linking of a external library, which was never done for any port before.

Static Linking

The *gc* compiler comes with a custom linker which is a modified version of the Plan9 linker [11]. The linker is not only responsible for combining multiple object files into a single executable binary, but also translates Plan9 pseudo assembly instructions into executable code [24] and eliminates dead code.

The default linker is fairly limited in a sense that, besides support for Plan9 object files, it is not capable of understanding the whole complexity of the ELF format [9], which becomes relevant when linking against object files produced by gcc. To overcome this limitation, the go distribution provides a workaround which extensively employs the host compiler and allows the default linker to perform dynamic linking.

However, this workaround is not implemented for static linking. Luckily, the linker is able to take advantage of a custom host linker by processing only Go and Plan9 assembly object files and using the host linker for the final step of the linking. In terms of Go, such linking is referred as *external*. The outcome of the external linking is a single statically-linked executable.

Calling C Functions

In order to call a C function from Go which is needed within the RTS, one could leverage a foreign function interface (*FFI*) called *CGO*. However, CGO depends on the Go runtime. Therefore, due to the circular dependency we decided to avoid using CGO when invoking libbarrelfish from the runtime.

Because the chosen Go compiler suite does not follow SystemV ABI conventions, ABI translation should be done before calling a C function. For this purpose, we reuse *runtime·asmsysvicall6* from the Solaris port. This helper function adjusts processor registers on entering and exiting a foreign function.

Each foreign function call requires a stack frame. It is not possible to reuse the stack frame of a regular Go routine because of the possibility to break the split stacks mechanism. The mechanism might request the OS to allocate a new chunk of memory for a frame and, subsequently, such request will be translated into a foreign function call because of dependencies such as memory allocation. Instead, we switch to a frame of an OS-spawned thread which does not rely on the contiguous stack mechanism. The switch is handled by the *asmcgocall* function which is used within CGO.

Because the contiguous stack mechanism for foreign functions is disabled, it is possible to overflow a stack without a notice. After experiencing consequences of the overflow during the porting process a few times, we tested

out the port with different stack sizes. It turned out that the stack does not overflow when its size is at least 8KB.

The other issue which has to be resolved when handling external C functions is blocking. Some functions might block, which means that the scheduler of Go will be blocked. However, in most cases it makes sense to schedule another goroutine while waiting the blocked function to return. Go provides a special helpers for handling that: *entersyscallblock* and *exitsyscall*. The former helper takes away the scheduling context P from the calling thread and if needed spawns a new OS thread. However, the usage of these helpers might impose an additional overhead because of the handling of the scheduling context, so they might not be suitable for functions which block for relatively short period of time.

libgo

The majority of the required functionality listed in the beginning of this section is provided either by libbarrelfish or by libposixcompat - a small POSIX [2] compatibility layer for Barrelfish. In addition, we introduced a *libgo* library which implements the missing functionality. Also, it wraps required functions from the former libraries into API functions.

The Go runtime and Barrelfish keep track of separate system states. The states are not shared and they can be implicitly synchronized when the runtime invokes an OS function. Therefore, implementation of some libgo functions requires special attention. For example, when the runtime asks the OS to allocate a given amount of memory at the beginning of a provided virtual address (*VA*), the OS should not map more memory than asked into the virtual address space. Otherwise, the state which tracks regions of allocated VA might diverge which can introduce inconsistency related bugs such as mapping the same VA region twice.

Initialization

The entry point for each Go application running on the Barrelfish OS is the same as for regular Barrelfish applications: *barrelfish_init_disabled*. The function sets up a domain and later on transfers the control to a *_rt0_amd64_barrelfish* function which is exported under the *_main* symbol. The function initializes the Go runtime by allocating some amount of memory which is used for malloc's pool, setting up the garbage collector and finally starting a scheduler. The scheduler starts the first Go routine which calls the main function of a user program.

3.1.3 Running Goroutines

As it is mentioned in Section 2.2, in Barrelfish, the smallest unit the CPU driver can schedule is a dispatcher. Upon invocation of upcall, the dispatcher schedules a thread. A default threading implementation is provided by libbarrelfish.

The Go runtime utilizes threads to schedule goroutines. Each thread can subsequently run multiple goroutines. By default, a goroutine is not pinned to a particular thread, therefore it can be run by multiple threads during its lifespan.

For the port, we rely on the libbarrelfish threading implementation. The reason for the choice is that it is not possible to introduce a new threading implementation without rewriting a significant part of libbarrelfish, because the current implementation is not modular enough and the threading is widely used by many modules within the library [46].

The Go runtime could benefit from custom threading, because the current implementation is not aware of any context of the runtime. For example, a throughput of a concurrent Go application could increase, if the dispatcher would schedule a thread which runs a receiving goroutine immediately after the thread has finished executing a goroutine which sent a message.

Figure 3.1 gives a high level overview of the Go application (*app.go*) running on Barrelfish. The application is running within the regular Barrelfish domain which consists of a single dispatcher. The Go dispatcher has established LMP channel with its local monitor. Also, in the given example, there exists the UMP channel to memory server (*mem_serv*) running on the other core (*core-0*). The server is in charge of memory allocations. The application embeds the run-time which consists of two Barrelfish threads (*M0* and *M1*) scheduled by the dispatcher. The M0 thread runs the *G2* goroutine which executes a blocking operation wrapped into entersyscallblock. The blocking results into creating the M1 thread and releasing the scheduler context (*P0*) which is taken by the newly created thread. After finishing executing G2, the M0 thread will wait until P0 gets released. The scheduler context includes a run-queue containing goroutines (*G1* and *G0*). The M1 thread executes the *G3* routine. After finishing executing it, the thread will schedule another routine from the run-queue. In the given example, there exists only one scheduling context. The number of contexts is controlled via *GOMAXPROCS* configuration parameter.

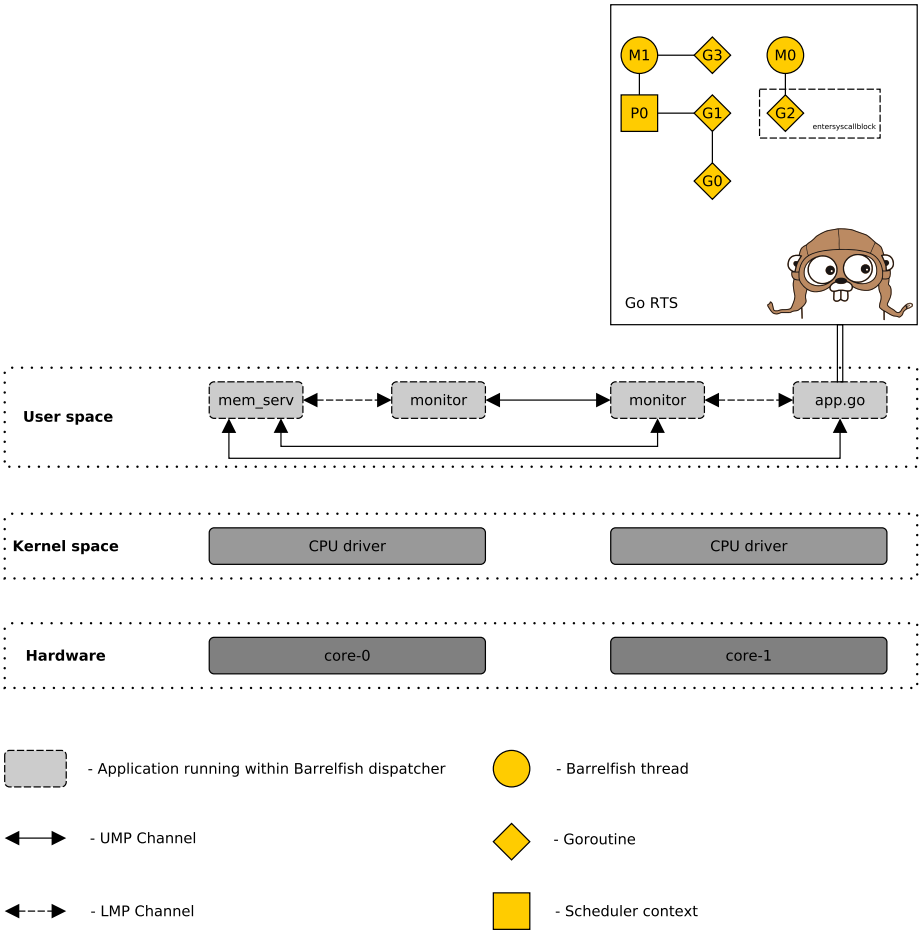


Figure 3.1: The figure shows the Go application (*app.go*) running on Barrelfish. The gopher image is taken from `http://golang.org`.

3.2 Channel Integration

The next step to make Go useful in the context of Barrelfish is to integrate Go channels with Flounder in such way, that the existing Barrelfish services could communicate with Go applications over the Go channels. Before starting to explore a space of possible integrations, we will compare message passing in terms of Barrelfish and Go and discuss possible difficulties for the integration.

3.2.1 Comparison of Message Passing in Barrelfish and Go

Even though Go and Barrelfish implement message passing by using channels, both implementations are different in many respects.

Types

Both Go and Barrelfish require channels to be typed. Typed channel ensures that only those messages which follow type rules can be sent via channel.

In Go, the type of a channel can be any valid primitive type, a particular interface or composition of both. Moreover, multiple type definitions per channel are not supported by Go. However, the empty interface (*interface*{}) is a valid channel type, which makes it possible to send any value via such a channel.

In Barrelfish, the channel type is declared by Flounder's DSL. Besides types which can be directly translated to primitive types of C and compositions of them, Flounder supports special types used to access Barrelfish capabilities. In contrast to Go, Flounder allows [1] us to send messages of different types over the same channel without losing type-safety properties.

Synchronousness

We say that a synchronous communication in the context of the message passing is the one that blocks a process till the completion of a operation, while asynchronous is the one which only initiates the operation and does not block a process [26].

Go supports synchronous communication. However, in the case of a buffered channel, sending does not block until the buffer has not been filled. Thus, we say that the buffered channel provides asynchronous communication, but only when it is not full. Using synchronous communication gives a useful property: both the sender and the receiver are synchronized and the sender knows at what state the receiver is.

[1] Only if a type for the message has been declared.

In contrast to Go, Barrelfish communication is only asynchronous, i.e. it does not block the sender in any case. Such communication is essential for distributed systems, because a guarantee of synchronization between sender and receiver is a difficult problem in such systems when compared to centralized ones [60] and it can deteriorate system performance [7].

Underlying Implementation

Because of its distributed nature, Barrelfish supports multiple interconnect drivers on which channels are built. However in Go, channels can only be used within the runtime. Therefore, they can be built by taking advantage of shared memory.

The underlying implementations and different system models determine the means of communication establishment. Establishment in Barrelfish requires relatively sophisticated mechanisms which involve monitor(s) and, quite often, services provided by the nameserver. Whilst in Go, the communication channel is created by the simple *make* statement which initializes a channel struct and allocates a buffer. The location of the channel is announced either by using global variables or by passing the channel as parameter in a statement which start a goroutine.

Next, the implementation of channels in Go follows an interesting idiom: channels are the first-class value and they can be send over other channels. Unfortunately, it is not possible with the current implementation of Flounder, although in Barrelfish, channels are represented as endpoint capabilities which can be passed over channels.

Operations

Barrelfish's send and receive operations have different levels of granularity in terms of error reporting in comparison to Go. In Barrelfish, sending might return an error, e.g., when the Flounder binding's outgoing buffer or the incoming buffer of a receiver are full. Go does not provide such feature. Instead, sending might cause a run-time panic, but only if a goroutine tries to send to a closed channel[2]. It is worth mentioning, that panics in Go are recoverable, therefore such failures can be handled without restarting the RTS.

Next, Barrelfish does not provide an explicit receive operator. As a substitute, receiving is handled in event-driven model [5] by executing the `event_dispatch` function on a waitset linked with the channel. Each message type is associated with a receive handler which is executed by the function. In spite the fact that such model can provide advantages in terms of

[2]Receiving from the closed channel always returns the zero value.

performance, it might make programs more difficult to write and to maintain [33].

For situations when the sender cannot make any progress without receiving a response message, Flounder provides RPC-style calls in the form of convenient wrappers.

3.2.2 Design Considerations

First of all, the design decisions for channel integration are influenced by the following constraints:

- Changing the implementation and semantics of the Go channels in a non-backward-compatible way would result in dismissing the possibility of embracing the rich ecosystem of Go 3rd party libraries in Barrelfish.
- Changing Flounder in a non-backward-compatible way would imply that existing Barrelfish services would not be able to communicate without any modification.
- Go does not provide API for accessing channels from foreign functions. Also, accessing Go pointers[3] from foreign functions is not defined behaviour as it is discussed in [3].

We choose asynchronous message passing because of the distributed nature of Barrelfish. It means that sending over the integrated channel should not block, regardless whether a receiver is ready to receive. Making channels synchronous in a distributed setting might slow down a system [7].

Following this, we put an emphasis on the easy-to-use property of Go channels. Therefore, we are in favor of ruling out the event driven model from the messaging path. As a consequence of this decision, receiving is done by the receive expression of Go which is easier to use than ”stack-ripping” technique.

Finally, after examining the existing codebase of Barrelfish and surveying the boilerplate code used to handle transient errors, we decided to hide such errors from the user and instead, handle them internally. A motivation for this decision is that in almost all cases, a transient error is solved by a registering continuation which will eventually send a message.

3.2.3 Implementation

The following section gives an insight into the implementation of the channels integration. First, we look into the details of the connection establishment. Afterwards, the send and receive operations will be presented.

[3] In Go, channel's value is a pointer to a struct.

For sake of clarity, we base our explanation on a Flounder interface shown in Figure 3.2. The interface is used to generate the Go package called `flounder/example` and the C module `go_flounder_example`. Both artifacts and the Go package `runtime/flounder` form the foundation for the integration of the particular interface. The source code of the artifacts can be found in Appendix A.

```
1 message ping(uint8 id, string msg);
2 message pong(uint8 id, string msg);
```

Figure 3.2: `example.if`: Flounder interface used in explaining the implementation details of the integration.

Before diving into the details, the following concepts should be explained:

- *Connection identifier*. Each connection is identified by an atomically incremented integer value. The value exists within the scope of the Go package and C module. In our example it is `flounder/example` and `go_flounder_example`.
- *Connection registry*. A hashmap used to map the connection identifier to attributes of the connection such as Go receive and send channels, deserialization functions and so on.
- *Default waitset*. A waitset which is associated with events triggering the Flounder export, bind and connect callbacks.
- *Non-default waitset*. A waitset which is associated with receive handlers.
- *EventDispatch goroutines*. For each Flounder interface, the runtime starts two goroutines which call `event_dispatch` for the default and the non-default waitset respectively. The latter goroutine handles connection establishment and receiving of messages.
- *Event list*. Connection establishment via the Flounder connect callback and message retrieval is handled by the foreign C functions. The C functions are not able to send to a Go channel. Therefore, the events are stored in the event list which is accessible by Go and C parts. The EventDispatch goroutine checks the list each time after executing `event_dispatch`.

An example of a client and a server running in separate domains and communicating via Go channels is shown in Figure 3.3. The Figure depicts communication between the client application (`domain-1`) and the server application(`domain-0`). The solid arrows denote communication over Go channels, while the dashed arrow shows communication by manually trig-

gering an event on Barrelfish's waitset. The communication is based on the `example.if` interface. The acceptor goroutine of the server is waiting for new connections on the `acceptChan` channel. After the new connection is established, the acceptor spawns the worker goroutine which is in charge of handling client requests. The worker goroutines on both sides send messages to the sender goroutines which call the Flounder send subroutines. The receiving is handled by the EventDispatch goroutines which after executing receive callbacks, deserialize messages and send them to the corresponding receive channels.

Connection Establishment

As it is mentioned in 2.2.2, during the connection establishment in Flounder, one part acts as a server and the other as a client.

Server Side

Export In order to accept a connection, the server should first export a service. The export is done by calling `example.Export(serviceName string)`.

The function does the following:

1. Creates a channel for receiving notifications about incoming connections.
2. Creates a connection identifier.
3. Creates a Go closure to be called when a new connection arrives. The closure is aware of the channel.
4. Registers the connection identifier and closure in the connection registry.
5. Calls a foreign function `go_flounder_example_export` which executes the Flounder export function `example_export`. As a state data, the connection identifier is passed to the function.

Once the local monitor allocates an iref for the server, the `export_cb` callback is executed. The callback associates the iref with the service name by registering the pair within the nameserver.

The initial Go export function returns the channel.

Accept Upon arrival of a new connection request, the following sequence of events happens:

1. The `connect_cb` callback function is executed. The function creates a new connection identifier and later on, it appends a new flounder

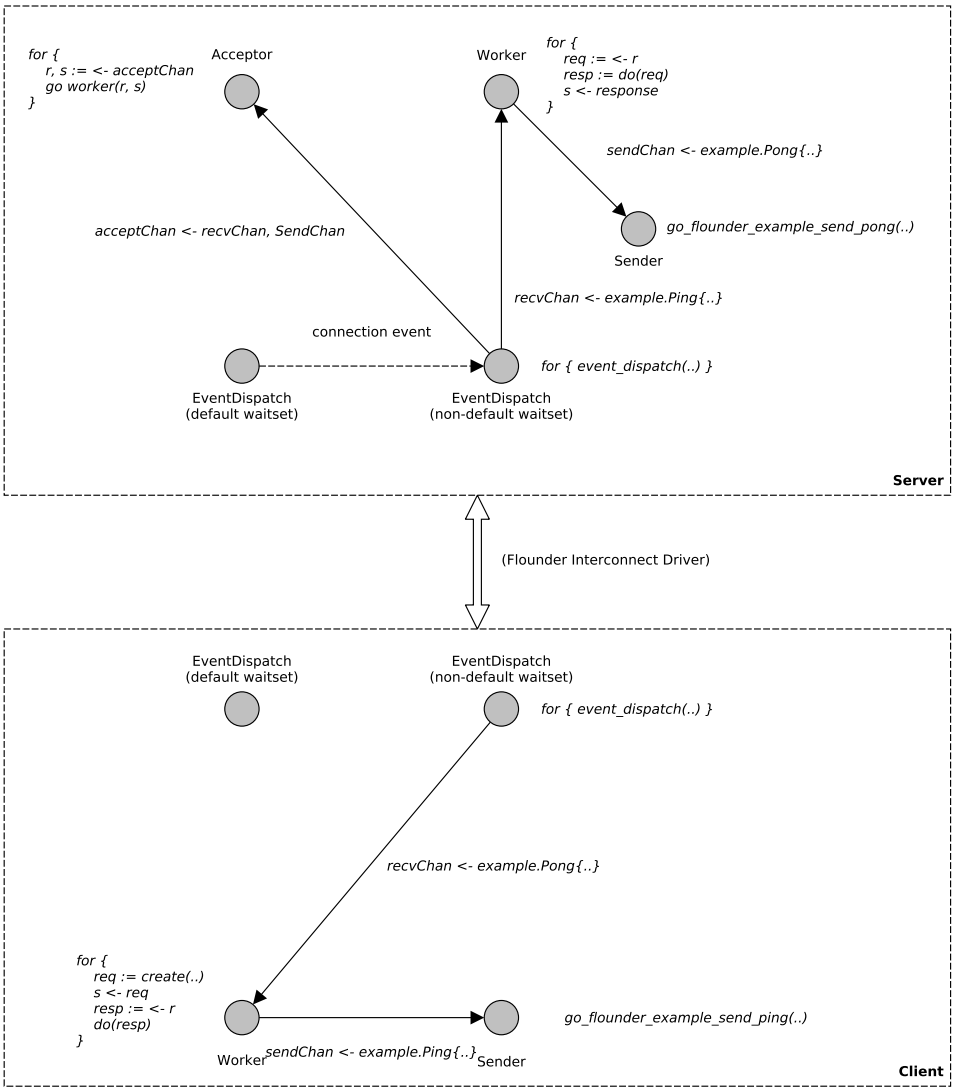


Figure 3.3: An example of a client and a server (in Go) which communicates over a Barrelfish's channel.

event (`EVENT_TYPE_CONNECT`) to the event list. The exported connection id and the newly created connection id are part of the event.

2. The event is processed by the `EventDispatch` goroutine. The routine executes the accept closure created in the export step.

3. The closure creates two channels: one for receiving and one for sending. Afterwards, it registers the new connection id and the receive channel in the connection registry. Following this, the closure spawns

a new goroutine which listens on the send channel. Finally, the pair of channels are sent to the accept channel.

The server gets notified about the new connection by executing the receive statement on the accept channel. The received values are the send and the receive channels.

Client Side The connection initialization from the client side is started by the `example.Connect(serviceName string)` function.

This function does the following:

1. Creates two channels (for sending and receiving).
2. Creates a connection identifier.
3. Registers the identifier together with the receive channel in the registry.
4. Calls a foreign function `go_flounder_example_connect`. The function returns a reference containing a pointer to a Flounder binding of the connection.
5. Creates a goroutine for sending messages. The parameters passed to the routine are the returned reference and the send channel. The latter is used by the routine for receiving send requests.

The foreign function does the serviceName to iref resolution by calling the nameservice. Afterwards, it calls the Flounder bind function `example_bind` which requests the Flounder connection establishment. The parameter passed to the function is the connection id. The function returns once the binding callback `bind_cb` is executed.

The return value of the connect API function is the pair of channels.

Sending

Sending a message is done via the regular Go send statement. The channel which is used for sending is of the `flounder.Serializable` type. The type is an interface which requires implementing the `Serialize` function. The function is used to serialize a message between Go and C types.

For each connection, there exists a goroutine which is listening on the send channel. The routine does the following:

1. Accepts the sent messages.
2. Serializes the message to C types.
3. Executes a foreign function which calls Flounder' send subroutine. E.g., `ping` message is sent by the `go_flounder_example_send_ping` foreign function.

Each message is stored in a Go struct. In the case of the `ping` message, a declaration of the struct representing the message is shown in Figure 3.4. The struct is serialized to a C struct shown in Figure 3.5. Memory for the latter is allocated by Barrelfish's `malloc` because of the constraint mentioned in 3.2.2 which says that no Go pointer can be accessed by external C functions.

In the case of the Flounder send function returning a transient error (`FLOUNDER_ERR_TX_BUSY`), the foreign function registers a continuation which should eventually deliver the message.

```
1 type Ping struct {
2     Id  uint8
3     Msg string
4 }
```

Figure 3.4: `example.Ping`: the Go struct used to represent the `ping` message.

```
1 struct params_ping {
2     uint8_t id;
3     char *msg;
4 }
```

Figure 3.5: The equivalent of the `example.Ping` struct in C.

Receiving

The receiving of Flounder messages is mostly handled by the EventDispatch goroutine. The routine calls `event_dispatch` on the non-default waitset and after the function returns, checks the event list for new events.

The receive event contains the connection id and a message type identifier used to select a function for deserialization of the message. The deserialized message is sent to the receive channel which is identified by the connection id. The channel is of the `interface{}` type, therefore messages of different type can be sent over such a channel.

3.2.4 Discussion

Firstly, the presented integration gives Go applications the ability to communicate with Barrelfish applications over the Go channels. In addition, communication among Go applications which run in separate Barrelfish domains becomes feasible. In both scenarios, no modifications to the existing Flounder and Go channel mechanisms are required. Also, the implementa-

tion ensures the communication to be asynchronous in a sense that sending does not block if a receiver is not ready to receive.

Secondly, the presented approach requires a few stub functions and declarations both in Go and C. This can be handled by a special purpose translator which takes the Flounder interface as a source and generates the required artifacts.

Thirdly, the given approach gives us an interesting property: the send and receive channels can be passed among goroutines over channels within the runtime boundaries. Therefore, we can implement different architectures for handling requests, e.g., worker pools.

Fourthly, the integration poses a few scalability-related issues. First of all, connection establishment and receiving is handled by the two event dispatching goroutines which can become a bottleneck. Next, all channels are buffered. However, the buffers are finite. Therefore, in the case of any buffer reaching its capacity, the EventDispatch goroutine might stall because of the blocking send behaviour. In addition, it makes the system not fully asynchronous.

Fifthly, due to the lack of union type in Go, the implementation loses some type-safety guarantees as there can be multiple types of message sent over the same channel.

Finally, the approach does not allow explicitly returning any error in the case when sending fails. One solution is to panic the sender goroutine and close the send channel. However, the original sender gets notified about the failure only after issuing the second send request. One could solve the problem by introducing additional channel for message delivery notifications. However, such solution would defeat our easy-to-use approach.

3.3 Implementing OS services in Go

After porting Go to Barrelfish and integrating Go channels with Flounder, we are able to create fully functioning Barrelfish applications in Go. In this section we will give a short overview of main considerations when building such applications. In addition, we will present a nameservice application which serves as a proof of concept for our port. Also, it could be used to replace the existing nameservice of Barrelfish.

3.3.1 Foreign Function Interface

As it is mentioned in Section 2.2, Barrelfish follows the library OS architecture and accordingly, a lot of functionality is exported in a form of libraries written in C. Thus, one of the main challenges when implementing applications for Barrelfish in Go is accessing such libraries.

One way to make the libraries accessible from Go is to rewrite them in Go. However, such an approach would require tremendous effort and in addition, it would require extra work for staying in sync with the original libraries.

Next, we could reuse the same approach as we did in the run-time integration: explicitly importing functions via static linking and wrapping them into ABI compatibility helpers. However, this approach burdens application development effort by requiring boilerplate. Also, public data types have to be redeclared and instances of them have to be manually serialized before calling any C function.

Luckily, Go provides a foreign function interface (*FFI*) to access C code which is called CGO. CGO allows us to call C functions and to make use of C data types. One disadvantage when using CGO is that all C function calls are considered as blocking ones and as it will be seen in Section 4.3.3, it might have negative implications on performance for some applications.

However, in spite of the disadvantage, we decided to extend[4] CGO in a way that it would work with Barrelfish and to use it as a primary mean for working with C code. The extension required a relatively small amount of work: creating stub functions, adding memory allocation functions and reusing Barrelfish threads for CGO calls.

Figure 3.6 shows a CGO example which allocates memory by using `frame_alloc` from *libbarrelfish*. It is woth mentioning that the provided snippet has undefined behaviour: rules for passing pointers of Go values to C has not been defined yet [3]. However, with the current version of Go, the example is functioning as expected.

[4] At the time of writing, CGO for Barrelfish is not integrated with the Go build system. Thus, some manual steps are required when building CGO applications.

```
/*
#cgo LDFLAGS: -L/barrelfish-build/x86_64/lib -lbarrelfish
...
#cgo CFLAGS: -I/barrelfish-build/x86_64/include
...

#include <barrelfish/barrelfish.h>
*/
import "C"

func main() {
    var frame C.struct_capref
    var err C.errval_t
    var size C.size_t

    err = C.frame_alloc(&frame, 64, &size)
    if C.err_no(err) == C.SYS_ERR_OK {
        fmt.Println("allocated:", size)
    } else {
        panic("frame_alloc")
    }
}
```

Figure 3.6: A snippet which uses CGO to access Barrelfish functions, declarations and macros.

3.3.2 Example Application: Nameservice

We have implemented the nameservice application as a replacement for the nameservice functionality provided by the System Knowledge Base (*SKB*) system service. The main responsibility of the nameservice is to handle translations between user-friendly names and iref's. As a backend, SKB uses a coordination service called Octopus [62].

The nameservice in Go implements a subset of `octopus.if` Flounder interface which includes `set`, `get` and `wait_for` RPC calls. However, our implementation does not support query language provided by Octopus. As a workaround for queries which require counting name instances we introduced the
`get_count` RPC call. The source code of the implementation is listed in Appendix B.

The underlying implementation of our nameservice relies on a hashtable which access is protected by a read-write lock. The lock is needed, because the a new hander goroutine is spawned per each connection which might access the map. In addition, we keep a list of pending requests to which the nameservice has to respond once a particular name has been registered.

The main advantage of our approach is the size of implementation: 102 lines of code excluding the Flounder generated stubs, comments and empty

lines. Also, its design encourages each client goroutine to have a separate connection, i.e. the existing nameservice client assumes a single connection per application. The assumption leads to a problem when multiple threads within a single application tries to query the nameservice over the same connection and they fail because of a RPC state of the connection. The state is ”in-progress” which means that no other concurrent request can be served.

Chapter 4

Evaluation

In this chapter we present series of benchmarks we performed to evaluate the port. In Section 4.1 we demonstrate macrobenchmarks based on the LevelDB database [30]. Later on, in Section 4.2 we present microbenchmarks which evaluate concurrent aspects of a Go application. Finally, in Section 4.3 we evaluate the channel integration in terms of performance characteristics.

The benchmarks presented bellow try to answer the following question:

- Does our port achieve the similar performance characteristics as the Linux port in different types of applications ?
- How big is an overhead for the integration of Flounder and the Go channels?
- What parts of the messaging path of the integrated channels take most of the time? Is it possible to eliminate or optimize them ?
- How slow is the integrated channels mechanism when compared to the Barrelfish messaging in C ?

4.1 LevelDB

In order to have an overall understanding about performance characteristics of the port, we perform macrobenchmarks based on a larger system which could be used in production and compare results against results of the same benchmarks performed on the Linux operating system.

Our choice for the system was limited, because many systems we considered follow the client-server model and thus, they can be accessed only via network. As is stated in [61], the current network stack of Barrelfish does not perform well and a performance of such systems running on Barrelfish would be most likely influenced by this fact. Therefore, the most

Suite	Description
fillseq	write N key-value pairs in sequential key order
fillrandom	write N key-value pairs in random key order
overwrite	overwrite N key-value pairs in random key order
batchwrite	write N pairs in multiple batches in sequential key order
readseq	read N pairs in sequential order
readrandom	read N pairs in random order

Figure 4.1: The ported LevelDB benchmark suites which we used in evaluation.

reasonable choice is an embeddable database. However, the majority of embeddable databases are optimized for persistency and block storage which usage we want to avoid in Barrelfish, because it might pose a botleneck for benchmarks [51] as well. Thus, the search space was narrowed down to in-memory embeddable databases.

After investigating multiple choices, we decided to use an alternative implementation of the LevelDB database called *goleveldb* [31]. As the name suggests, the alternative version is implemented in Go, while the former version of the database is written in C++. The original database acts as a persistent key-value store. However, the implementation in Go brings to the table a memory-only mode. The database stores key-value pairs lexicographically sorted by keys which implies good performance characteristics for lexicographically sequential data access patterns. Internally, pairs are stored in a form of the Sorted String Table (*SSTable*) data structure [32] which requires storing an additional table for key-offset pairs. An offset points to a value which is associated with a key in SSTable.

The database supports data compression by using the Snappy compression library [50], hence its memory footprint can be reduced.

It is worth mentioning, that the benchmark does not stress our channel integration. Instead, it tests the initial port which is described in Section 3.1.

4.1.1 Suites

The original implementation of LevelDB includes a number of benchmark suites which are used when comparing LevelDB to similar data stores [52]. For our benchmark purpose, we decided to be compatible with the suites and therefore we ported the benchmarks to Go. Table 4.1 contains description of the ported benchmark suites.

4.1.2 Setup

First of all, we used the following configuration parameters for the database and the suites:

- Key size is equal to 16 bytes as it used in the original benchmarks.
- Value is a 100 bytes long compressable random string with a compression rate equal to 0.5. In addition, randomly generated sequence is identical on both operating systems. Same applies to random order of requests. All values are the same as in the original benchmarks.
- N is equal to 1 000 000 per benchmark suite.
- For read benchmarks we populate database with 1 000 000 entries.
- In all benchmarks batch size is equal to one except for the batchwrite suite in which the size is set to 1 000.
- All requests are made from a single goroutine, because the runtime can utilize only a single core on the Barrelfish OS.
- GOMAXPROCS is set to one, because we use only one goroutine for issuing requests.
- When the garbage collector is disabled, we force garbage collection in-between each run to avoid running out memory.

Next, we ran benchmarks on the *release2015-07-14* [1] version of Barrelfish and on the *Ubuntu 14.04 LTS (Trusty)* distribution of Linux with kernel 3.13.0. For the Go version, we use *go1.4beta1-1732-g9b69196* [2] on both operating systems, because the port is based on this version.

The benchmarks were run on *Intel Ivy-Bridge* and *Intel Haswell* machines. The configuration of the machines is presented in Table 4.2.

It is necessary to mention that *Turbo Boost* was disabled on both machines to avoid fluctuations in CPU frequency. In addition, the CPU affinity of all benchmarks was set to one, which means that all benchmarks were performed on the first core of the machines.

The duration of each operation within the benchmark suites was measured by using the *time.Now* function. The underlying implementation of the function is based on the *gettimeofday* and *clock_gettime* functions in Barrelfish and Linux, respectively. The overhead of *time.Now* is minimal and is roughly 100 CPU cycles on both operating systems which corresponds to 0.4 micro seconds on a 2.5 GHz processor.

[1] `http://git.barrelfish.org/?p=barrelfish;a=shortlog;h=refs/tags/release2015-07-14`

[2] "g9b69196" corresponds to a commit id of `https://github.com/golang/go`.

Microarchitecture	CPU Model	Sockets x Cores
Intel Ivy Bridge-EP	2.50 GHz Intel Xeon E5-2670 v2	2 x 10
Intel Haswell	3.40 GHz Intel Xeon E3-1245 v3	1 x 4

Figure 4.2: Characteristics of machines used in evaluations.

4.1.3 Results

The first results of benchmarks conducted on Intel Ivy-Bridge and Intel Haswell are presented in Figure 4.3 and Figure 4.4, respectively.

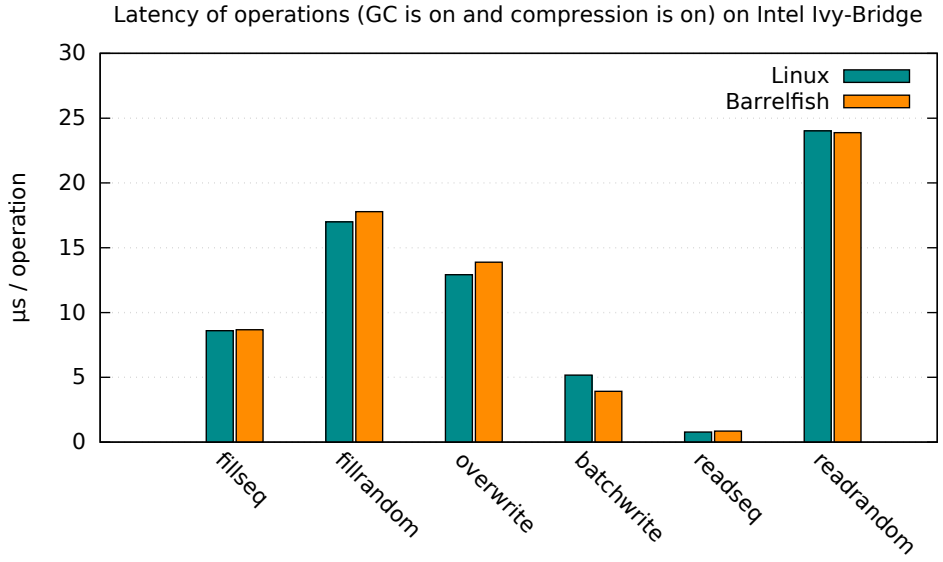
As can be seen, on the Ivy-Bridge machine the performance characteristics are similar between goleveldb running on Linux and Barrelfish. The same holds for results conducted on the Intel Haswell machine. However, the *batchwrite* benchmark is slightly faster on Barrelfish, while the *overwrite* benchmark is faster on Linux. To clarify the differences, we could take advantage of Intel Performance Monitoring Unit. However, at the time of writing, there did not exist any library in Go which could query hardware performance counters.

Looking into profiles generated by the Go profiler on Linux[3] system which can be seen in Figure 4.5, we can notice that data compression operation (`snappy.Encode`) is taking most of the time. Disabling it should make GC-related operations (`runtime.scanobject` and `runtime.writebarrierptr`) more frequent. Thus, any differences in GC performance between OSes would have a bigger influence on the results. Figure 4.6 shows that disabling made benchmarks to run faster and there is a small change in the `readrandom` benchmark result: this operation is slightly faster on Linux.

Next, all operations inherit high standard deviation. One reason for that is the garbage collector which stops the world. Figure 4.7 shows results when the GC is disabled. Disabling the GC reduced the standard deviation when compared to Figure 4.4. However, it stayed higher than average for each operation which confirms a fact that high standard deviation is inherit to LevelDB [29]. Figure 4.8 shows results when latencies of outliers which are above 99th percentile are filtered out.

In addition, we did not test the suites with the recent version of Go. The recent version includes a concurrent GC which ensures shorter stop-the-world pauses [36].

[3]We could not generate profiles on Barrelfish, because the Go profiler heavily relies on UNIX signals and therefore porting the profiler to Barrelfish would require a significant amount of work.



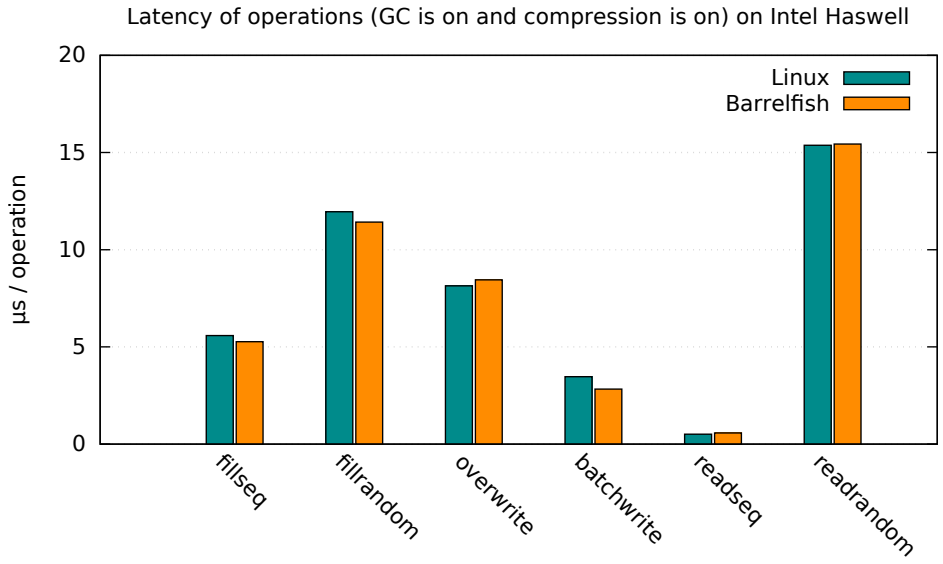
Suite	Mean	Std	99%-ile
fillseq (linux)	8.60	625.62	8.46
fillseq (barrelfish)	8.67	726.23	8.00
fillrandom (linux)	16.99	1510.89	11.05
fillrandom (barrelfish)	17.78	2075.12	10.00
overwrite (linux)	12.92	1796.95	8.57
overwrite (barrelfish)	13.89	1237.65	9.00
batchwrite (linux)	5.18	657.82	0.45
batchwrite (barrelfish)	3.92	492.67	1.00
readseq (linux)	0.77	1.92	11.25
readseq (barrelfish)	0.85	1.98	12.00
readrandom (linux)	24.02	1421.98	29.40
readrandom (barrelfish)	23.88	1337.56	29.00

Figure 4.3: Goleveldb benchmark results on Intel Ivy-Bridge measured in micro seconds.

4.2 Concurrency

To evaluate the performance characteristics of concurrent Go programs, we designed a special benchmark which extensively employs goroutines and channels. The benchmark starts multiple goroutines which form a ring. Each goroutine is numbered. Upon receiving a message, a goroutine sends it to its neighbour, whose ID is equal to the sender's ID incremented by one, via a channel connecting them. Figure 4.9 illustrates the described ring.

The benchmark measures the round-trip time of a message which is sent by the first goroutine. Because of the topology, the same goroutine receives



Suite	Mean	Std	99%-ile
fillseq (linux)	5.59	470.65	4.59
fillseq (barrelfish)	5.27	465.84	5.00
fillrandom (linux)	11.95	1240.22	6.44
fillrandom (barrelfish)	11.42	1553.51	6.00
overwrite (linux)	8.15	1220.77	4.80
overwrite (barrelfish)	8.45	1214.47	5.00
batchwrite (linux)	3.47	450.60	0.30
batchwrite (barrelfish)	2.83	362.03	1.00
readseq (linux)	0.51	1.27	7.18
readseq (barrelfish)	0.58	1.31	7.00
readrandom (linux)	15.38	987.88	17.11
readrandom (barrelfish)	15.43	935.65	17.00

Figure 4.4: Goleveldb benchmark results on Intel Haswell measured in micro seconds.

the final message after it has traversed the whole ring. Therefore, the first goroutine is responsible for taking time measurements.

The benchmark evaluates not only sending and receiving of messages, but also scheduling of goroutines. It is due to the fact that messaging related operations involve scheduler activities such as unblocking and scheduling a goroutine which is e.g. blocked by the receive statement.

Percentage	Function
8.65	`snappy.Encode`
6.02	`runtime.heapBitsForObject`
6.02	`runtime.scanobject`
4.70	`runtime.writebarrierptr`
4.32	`runtime.selectgoImpl`

Figure 4.5: CPU profile for *fillrandom* when running on Linux on Intel Haswell machine. The first column shows a percentage of how many times a function appeared in a sample of the profile.

4.2.1 Setup

The benchmark uses the same machines and operating systems presented in Section 4.1.2.

The benchmark properties include:

- All channels are unbuffered, i.e., sending a message blocks a sender until a receiver is ready to receive it.
- The payload of the initial message is an integer whose value is equal to one. After each hop, the payload gets incremented by one, therefore the final message value is equal to the ring size.
- We run benchmark 100 times for each ring size. The number was empirically determined after figuring out that more than 100 runs do not improve standard deviation.
- GOMAXPROCS and the garbage collector settings are the same as in Section 4.1.2.

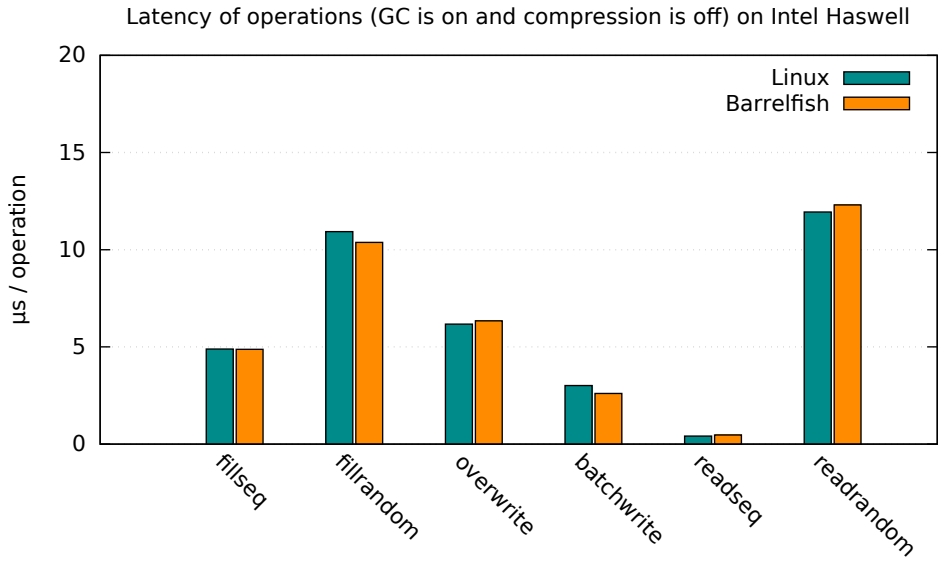
The time measurements are done by using the `time.Now` function.

4.2.2 Results

The benchmark results are shown in Figure 4.11. First of all, we note that the message latencies were lower on Linux than on Barrelfish when running on both machines.

Next, it can be seen from Figure 4.11, that the latency grows linearly when ring size increases. Nevertheless, in the beginning it does not hold for Barrelfish on either machine. In addition, the latency has a high standard deviation for that period on Barrelfish.

Looking into generated profiles on the Linux machine shown in Figure 4.10, we deduce that 20% of the whole run time the benchmark spent in garbage collection and 20% of the time in executing the receive statement.



Suite	Mean	Std	99%-ile
fillseq (linux)	4.89	421.96	4.66
fillseq (barrelfish)	4.87	414.68	5.00
fillrandom (linux)	10.93	1231.46	6.32
fillrandom (barrelfish)	10.38	1505.16	6.00
overwrite (linux)	6.17	433.32	4.64
overwrite (barrelfish)	6.34	737.11	5.00
batchwrite (linux)	3.01	432.14	0.29
batchwrite (barrelfish)	2.61	371.19	1.00
readseq (linux)	0.41	0.62	3.45
readseq (barrelfish)	0.47	0.74	4.00
readrandom (linux)	11.94	996.67	13.15
readrandom (barrelfish)	12.30	1003.52	13.00

Figure 4.6: Goleveldb benchmark results on Intel Haswell measured in micro seconds. Compression is disabled.

Disabling garbage collection fixed the problem with high standard deviation as it is shown in Figure 4.12. Therefore, we can conclude that the latency standard deviation spikes were due to pauses of GC.

Next, disabling GC had an effect on latencies which became roughly the same when running on the same machine. Thus, we can infer that the benchmark is sensitive to GC activities.

Suite	Mean	Std	99%-ile
fillseq (linux)	4.15	128.49	5.52
fillseq (barrelfish)	5.61	238.52	5.00
fillrandom (linux)	7.79	273.21	6.74
fillrandom (barrelfish)	9.92	910.33	6.00
overwrite (linux)	7.46	1031.85	4.90
overwrite (barrelfish)	7.40	964.70	5.00
batchwrite (linux)	2.18	142.85	0.28
batchwrite (barrelfish)	2.23	183.46	1.00
readseq (linux)	0.51	1.26	7.13
readseq (barrelfish)	0.57	1.31	7.00
readrandom (linux)	12.67	2.24	15.97
readrandom (barrelfish)	13.28	15.31	17.00

Figure 4.7: Goleveldb benchmark results on Intel Haswell machine measured in micro seconds. GC is disabled.

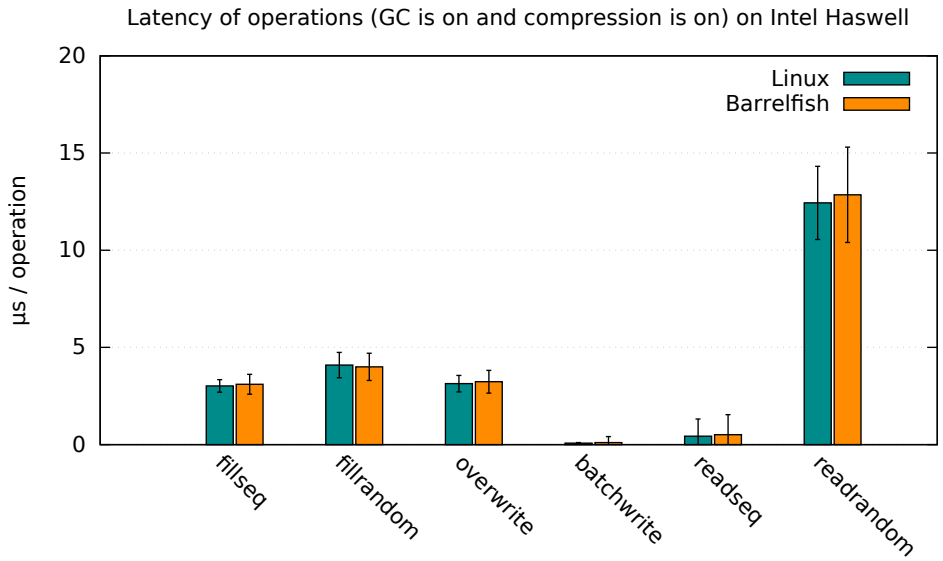


Figure 4.8: Goleveldb benchmark results on Intel Haswell measured in micro seconds when outliers are filtered out.

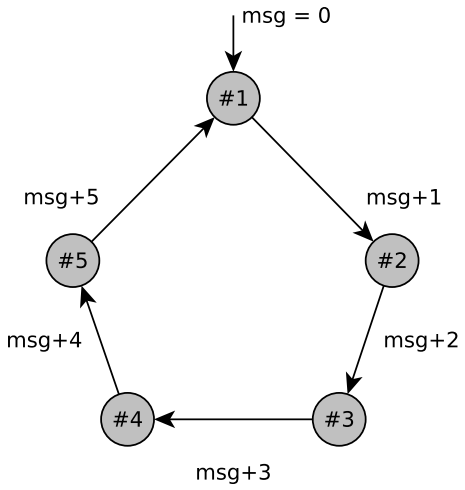


Figure 4.9: The topology of the ring when the number of goroutines is equal to five. The arrows denote channels used by neighbouring goroutines to communicate and direction of the communication.

Percentage	Function
47.25	`runtime.goexit`
34.34	`main.worker`
31.87	`runtime._System`
20.88	`runtime._GC`
20.33	`runtime.chanrecv1`
20.05	`runtime.chanrecv`
9.62	`runtime.backgroundgc`
9.62	`runtime.gc`
9.34	`runtime.scanobject`
9.07	`runtime.gcDrain`

Figure 4.10: CPU profile for the concurrency benchmark conducted on Linux machine and sorted by cumulative percentage. The first column shows a percentage of how many times a function appeared in a *stacktrace* of any thread during the profiling.

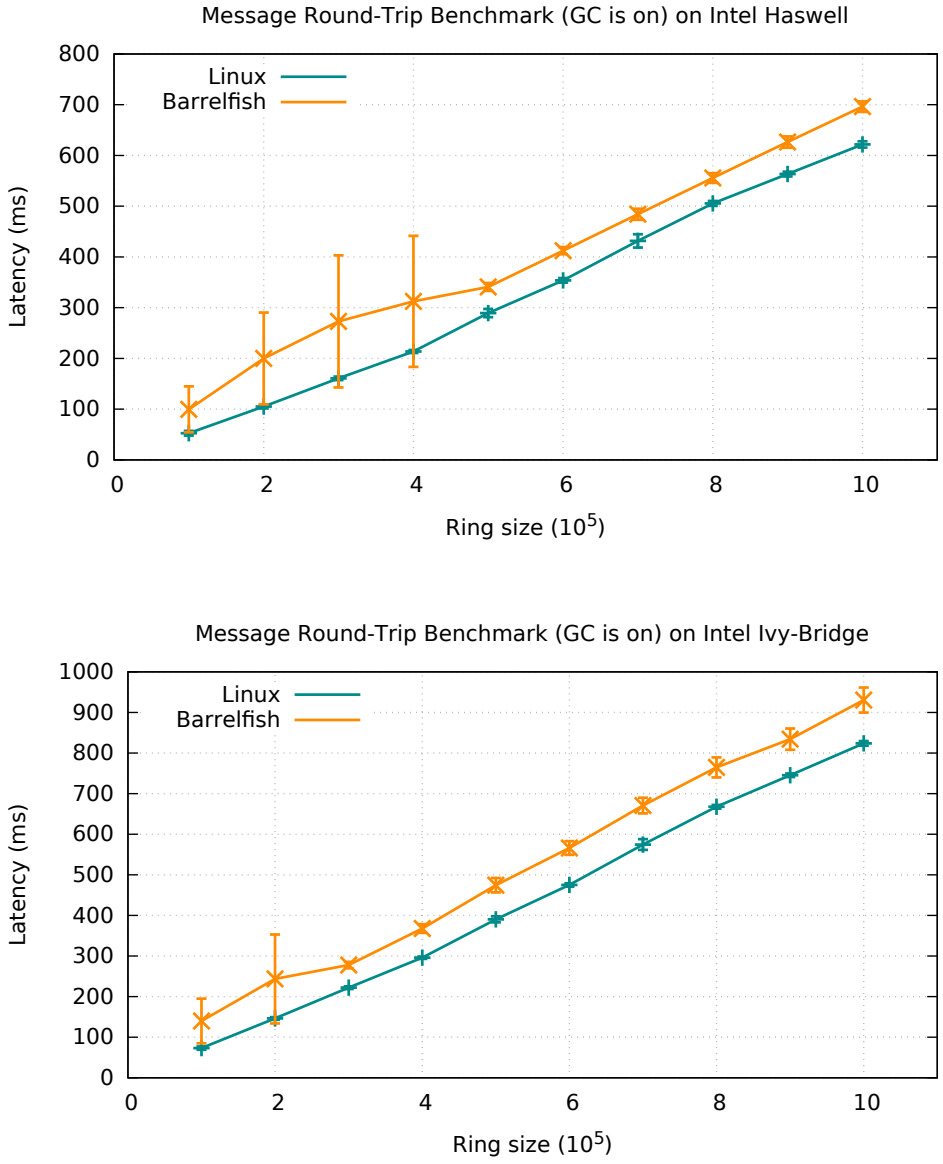


Figure 4.11: The concurrency benchmark results on Intel Ivy-Bridge and Intel Haswell. GC is enabled.

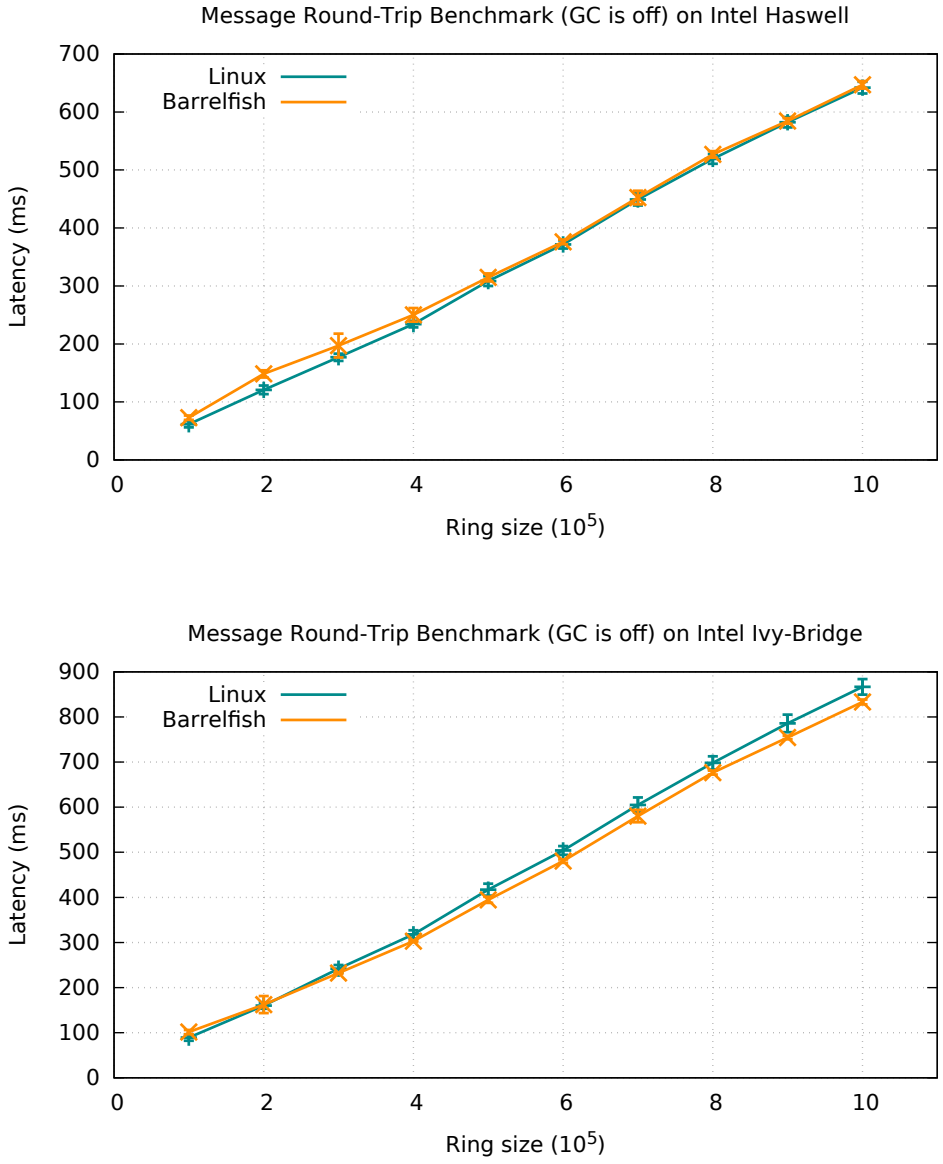


Figure 4.12: The concurrency benchmark results on Intel Ivy-Bridge and Intel Haswell. GC is disabled.

4.3 Channel Integration

To answer the performance related questions of the channel integration, we designed a benchmark. The benchmark consists of a single server application and multiple client applications. Each client application is sending a ping message to the server over Flounder channel and consequently the server responds with a pong message. The Flounder interface used for the communication is shown in Figure 4.13. A payload of the ping and pong messages consists of a client id and a message id. Both values are of 32 bits size, therefore they can fit into a single UMP packet.

```
1 message ping(uint32 client_id, uint32 msg_id);
2 message pong(uint32 client_id, uint32 msg_id);
```

Figure 4.13: `ping.if`: The flounder interface used in the channel integration benchmarks.

Next, we have implemented client and server side of the ping-pong applications both in Go and C. The C implementation serves for a baseline purpose in the performance evaluation.

A simplified version (excludes statements used for the timing and profiling measurements) of the ping-pong server in Go is presented in Figure 4.14. The server takes an advantage of the concurrency primitives of Go by spawning a goroutine for each connection. In addition, it relies on the Flounder channel integration described in Section 3.2.3. The server in C is shown in Figure 4.16. Unlike the implementation in Go, the C server handles all communication in a single thread. Both implementations present the idiomatic way to handle multiple connections both in Go and Barrelfish.

The client part of the ping-pong application for Go and C is shown in Figure 4.15 and Figure 4.17, respectively. For the benchmark we assume that any instance of the client application establishes only one connection to the server, which is usually the case for the Barrelfish applications. It is worth mentioning that Go client is able to communicate with the C server and vice-versa. The same applies to the C client and the Go server.

```go
func main() {
    acceptChan := goping.Export("msg_benchmark")
    for acceptPair := range acceptChan {
        go worker(acceptPair)
    }
}

func worker(p goping.AcceptPair) {
    for {
        req := (<-p.RecvChan).(goping.Ping)
        p.SendChan <- goping.Pong{req.Id, req.MsgId}
    }
}
```

Figure 4.14: The implementation of the ping-pong server in Go.

```go
func main() {
    sendChan, recvChan := goping.Connect("msg_benchmark")
    for i := 0; i < msgCount; i++ {
        sendChan <- goping.Ping{uint32(clientId), uint32(i)}
        <-recvChan
    }
}
```

Figure 4.15: The implementation of the ping-pong client in Go.

```
static void rx_ping(struct ping_binding *b,
                    uint32_t client_id, uint32_t msg_id) {
    errval_t err;
    err = b->tx_vtbl.pong(b, NOP_CONT, client_id, msg_id);
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err, "pong");
    }
}

static struct go_ping_rx_vtbl rx_vtbl = {
    .ping = rx_ping,
};

/* The implementation of the "connect_cb" and "export_cb" callbacks is omitted
for sake of brevity. */

int main(int argc, char *argv[]) {
    errval_t err;
    struct waitset *ws = get_default_waitset();
    err = ping_export(NULL, export_cb, connect_cb, ws,
                      IDC_EXPORT_FLAGS_DEFAULT);
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err, "ping_export");
    }
    while (true) {
        err = event_dispatch(ws);
        if (err_is_fail(err)) {
            USER_PANIC_ERR(err, "event_dispatch");
        }
    }
    return 0;
}
```

Figure 4.16: The implementation of the ping-pong server in C.

```
static void send(struct ping_binding *b) {
    errval_t err;
    if (cur_msg_id != msg_count) {
        err = b->tx_vtbl.ping(b, NOP_CONT, client_id, cur_msg_id);
        if (err_is_fail(err)) {
            USER_PANIC_ERR(err, "ping");
        }
        cur_msg_id++;
    }
}

static void rx_pong(struct ping_binding *b, uint32_t client_id, uint32_t
    msg_id) {
    send(b);
}

static struct go_ping_rx_vtbl rx_vtbl = {
    .pong = rx_pong,
};

static void bind_cb(void *st, errval_t err, struct ping_binding *b) {
    b->st = st;
    b->rx_vtbl = rx_vtbl;
    cur_msg_id = 0;
    send(b);
}

int main(int argc, char *argv[]) {
    errval_t err;
    struct waitset *ws = get_default_waitset();
    err = ping_bind(iref, bind_cb, NULL, ws, IDC_BIND_FLAGS_DEFAULT);
    if (err_is_fail(err)) {
        USER_PANIC_ERR(err, "ping_bind");
    }
    while (true) {
        err = event_dispatch(ws);
        if (err_is_fail(err)) {
            USER_PANIC_ERR(err, "event_dispatch");
        }
    }
    return 0;
}
```

Figure 4.17: The implementation of the ping-pong client in C.

4.3.1 Setup

The benchmark consists of a single server application running on the core-1 and one or more client applications. We decided not to run the server on the core-0, because that core runs other applications which interference might influence benchmark results. Each client application is running on a separate core starting with the core-2, i.e. the first client is running on the core-2, the second client on the core-2 and so on. As the consequence of such configuration, the UMP channels are used among applications.

All benchmarks were conducted on the Intel Ivy-Bridge machine described in Table 4.2. It is noteworthy that the machine consists of two NUMA nodes: core-0 to core-9 belong to the first, while core-10 to core-19 to the second. Therefore, all client applications running on the second node might experience higher latencies for message passing, because the server is running on the first node.

In all Go applications, we set GOMAXPROCS equal to one, because an instance of such applications does not span multiple cores.

During the benchmarks, the clients send $1,000,000$ of the ping messages all together. E.g., in the case of two clients, each client sends $500,000$ requests.

The timing measurements are done on the server side by using the `rdtscp` CPU instruction. The server measures how many cycles it takes to receive the predefined number of the ping requests. Each benchmark is run 20 times. After 20 iterations, we restart the benchmark and increase number of clients by one.

The synchronization among server and multiple clients was achieved by using distributed barriers of the `dist` library.

Finally, we run benchmarks with multiple configurations:

- Single server in Go and one or more client in Go.
- Single server in Go and one or more client in C.
- Single server in C and one or more client in Go.
- Single server in C and one or more client in C.

4.3.2 Results

First of all, we expect the performance of the ping-pong applications in which either server or client or both are in Go to be worse than the performance of the same system consisting of both server and clients implemented in C. This is due to the overhead introduced by the additional channels and goroutines used to make Flounder communication possible over the Go channels.

The actual benchmark results are presented in Figure 4.18. As it can be seen, our hypothesis proved to be correct: the ping-pong system consisting of the Go server and the single client in Go turns to be ≈ 77 (19.24 vs 0.24 seconds) times slower than the equivalent system which both parts are in C, and ≈ 44 (6.29 vs 0.13 seconds) times slower in the case of 6 clients.

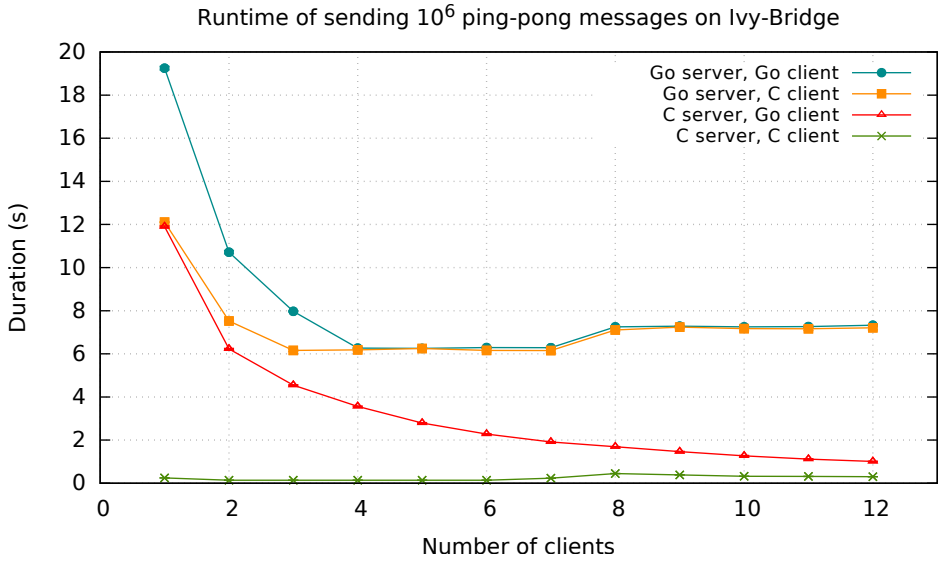


Figure 4.18: Runtime of the ping-pong benchmark when the number of clients increases. Each client sends $\frac{10^6}{number\_of\_clients}$ requests.

Next, the server in Go cannot be fully utilized unless the number of concurrent clients has reached three. This can be deduced from the *"Go server, C client"* plot, in which the minimum runtime is achieved with three C clients.

Furthermore, a single client in Go is not able to generate enough load which could make both servers to reach their maximum capacity. This claim can be backed by the fact that in the *"Go server, C client"* plot the best runtime is achieved with less clients than in the *"Go server, Go client"* plot (3 vs 4 clients). Also, in the *"C server, Go client"* we can see that the runtime decreases when the number of Go clients increases.

Finally, there is a runtime increase in all cases except for *"C Server , Go client"* when the number of clients reaches **8** and above. The increase is expected because of at some point the new clients will run on the different NUMA node than the server does. However, it should happen when the number of clients is equal to **9**.[4] Further benchmarks have showed, that the ping-pong

[4]The 9th client will run on core-10 which belongs to the second NUMA node.

application in the case of 8 clients achieves better performance when two of the clients run on the separate nodes than the rest.

4.3.3 Optimizations

After seeing the initial results, we took a step further to identify potential bottlenecks and possible improvements by breaking down the runtime of the benchmark in which both server and client implementations are in Go and the number of clients is equal to one.

The breakdown is shown in Figure 4.19. In addition, Figure 4.20 depicts a sequence diagram containing each individual part of the breakdown which includes:

- `C1`, `SD6`, `S2`, `CD6`: sending a message over Go channels between two goroutines.
- `C2`, `S3`: serializing a message from Go to C. The serialization makes a blocking call to Barrelfish' `malloc` function.
- `C3`, `S4`: entering a C function which will call a Flounder stub function to send a message over a channel of Barrelfish. The entering gives away the scheduling context.
- `C4`, `S5`: executing the Flounder stub function.
- `C5`, `S6`: freeing of a memory used for the serialization.
- `C6`, `S7`, `CD2`, `SD2`: exiting a C function and trying to reclaim the scheduling context.
- `CD1`, `SD1`: creating an event for message arrival in C which is passed over the global variables to Go.
- `CD3`, `SD3`: retrieving attributes from the connection registry for a received event.
- `CD4`, `SD4`: deserializing a message from C to Go.
- `CD5`, `SD5`: freeing a memory used for creating a message in C.

The `Flounder` part denotes percentage of the total elapsed time used to send the request and the response over the Flounder channel.

First of all, we can notice that operations responsible for memory allocation (`malloc`, `free`) happening in Go context (`C2, S3, CD5, SD5`) contribute to 19.58% of the total execution time. We make a hypothesis that the operations yields lower overhead than it appears on the messaging path. The explanation for higher overhead is that the operations are treated as blocking ones by the Go runtime system. Therefore, each operation involves handing off the scheduling context and after the operation returns, the thread executing

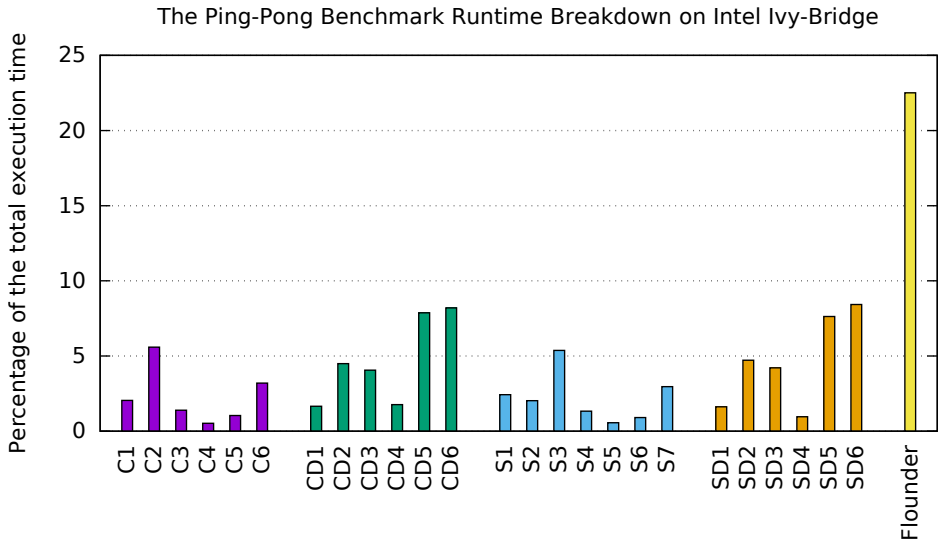


Figure 4.19: Runtime breakdown of the ping-pong benchmark when both server and client are implemented in Go and there is only one client. The breakdown is for a single request-response pair measured over 10^6 samples.

the goroutine tries to reclaim the context. The latter (`C6`, `S7`, `CD2`, `SD2`) contributes to 15.37% of the total.

Next, sending the deserialized message from the dispatcher goroutine to the client and the server goroutines (`CD6`, `SD6`) takes 16.63% of the total time. The reason for the relatively high fraction is that the receivers are blocked and sending does not schedule them immediately. Instead, the send statement puts them into the run-queue of the scheduling context allowing them to be run eventually.

Finally, sending and receiving over a channel established by Flounder contributes to 22.50% of the total execution time. The absolute value for our case is 4.93 μseconds, while in the benchmark presented in Section 4.3.2 the runtime for the single round-trip is equal to 0.24 μseconds (from the *"C server, C client"* case when there was one client). The difference can be explained by the fact that the runtime consists of more than one thread and the thread executing `event_dispatch` is not scheduled immediately.

After the analysis, we decided to improve the Flounder integration by eliminating multiple goroutines from the messaging path and therefore receiving the messages without using the Go channels. The simplified version of the optimized server is shown in Figure 4.22. The code closely resembles the way the Flounder communication is handled in Barrelfish applications. Also, it removes the need for the sender and `event_dispatch` goroutines.

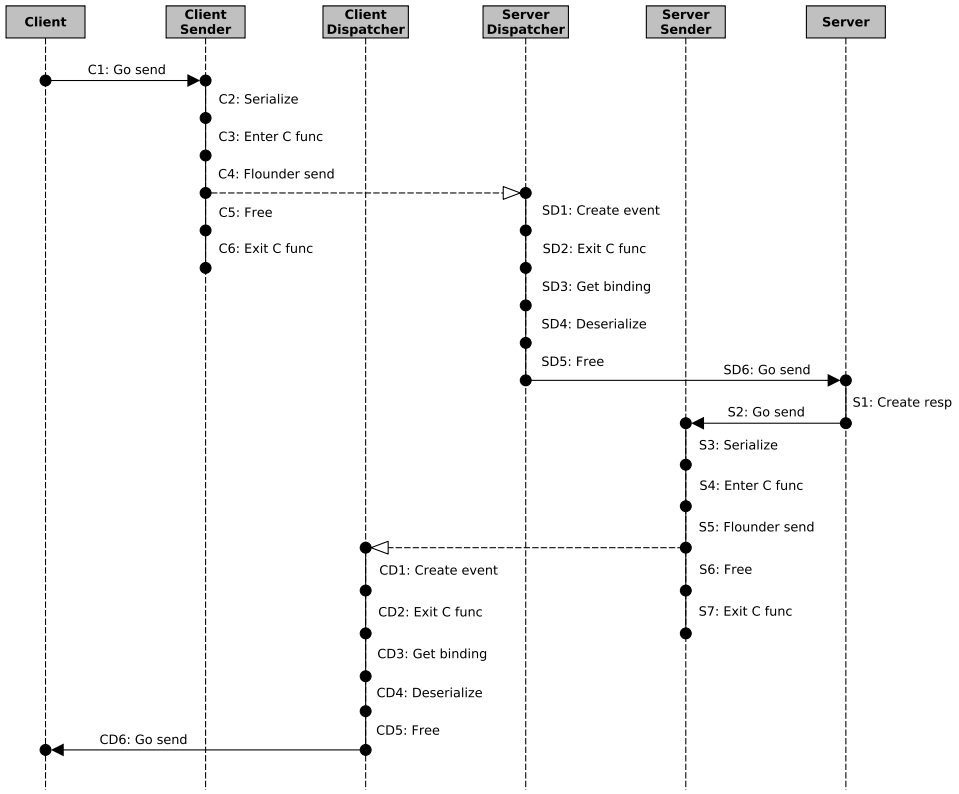


Figure 4.20: Sequence diagram for a single round-trip of a message traversing over the integrated channels. The dashed arrows indicate communication happening over Flounder.

The same modifications have been applied to the client side.

Consequently, we pinned the server and client to particular Barrelfish threads by using the `runtime.LockOSThread` function. The function ensures that no other goroutine can be scheduled on the pinned thread. Hence, it should provide lower latencies when scheduling the main goroutine.

In the end, we changed the blocking operations in a way that they do not hand off the scheduling context during their execution.

The performance of optimized versions is shown in Figure 4.21. The *"C server, C client"* and *"w/ Channels (Go server, Go client)"* plots are taken from Figure 4.18.

The *"w/o Channels (Go server, Go client)"* plot represents performance of the server and client which eliminates the additional routines and, as a consequence, the Go channels as it is shown in Figure 4.22. The *"w/o Channels, optimized (Go server, Go client)"* is conducted by using the just described client and server with the pinning and blocking optimizations enabled.

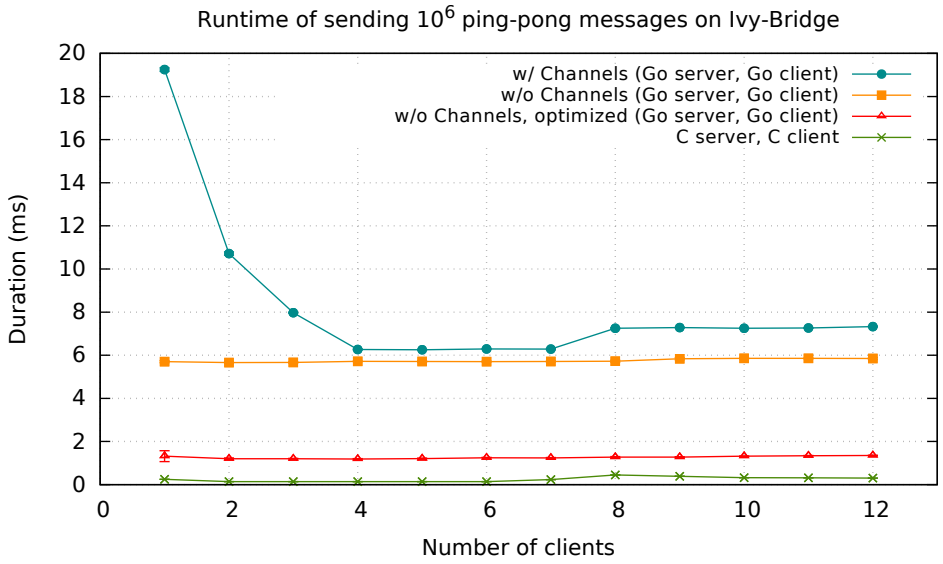


Figure 4.21: Runtime of the ping-pong benchmark when the optimizations are applied.

```
1  func acceptHandler(bid flounder.BindingId) {
2      // executed upon connection establishment
3  }
4
5  func receiveHandler(bid flounder.BindingId, ping goping.Ping{}) {
6      flounder.Send(bid, goping.Pong{ping.Id, ping.MsgId})
7  }
8
9  func main() {
10     goping.Export("go_bench", acceptHandler, receiveHandler)
11     for {
12         flounder.EventDispatch()
13     }
14 }
```

Figure 4.22: The optimized version of the ping-pong server in Go.

As it can be seen from the plots, the first version brought an improvement which allows the server to be fully utilize with a single client. However, the improvement of this version is relatively small which is $\approx$ 1.1 times faster (5.70 vs 6.29 seconds) in the case of 6 clients.

Next, the optimized version has the similar behaviour in terms of the utilization. In addition, this version has more significant impact on the performance: it makes the system to run $\approx$ 5 faster (1.24 vs 6.29 seconds) with 6 clients when compared to the original version and $\approx$ 8.9 times slower than

the C version.

It is worth mentioning that the new versions of the system experience a small increase in the runtime when the number of clients goes beyond 8. It is so, because new clients starting with the 9th client are running on the different NUMA node than the server does. Thus, it is expected that the message passing over UMP is slower in this particular situation.

Besides the mentioned changes, we could apply even more optimizations. For example, memory allocations by using `malloc` can be replaced with a slab allocator technique. However, we decided to keep the optimization out of the scope of the thesis.

Chapter 5

Conclusions and Future Work

5.1 Future Work

This section discusses further steps which can be taken in order to improve and to extend the Go port.

5.1.1 Improving Send Semantics

As it discussed in Section 3.2.4, the Flounder send operation over the Go channels does not properly propagate errors to the sender. As a solution to this problem, we suggest to introduce a special compiler directive in a form of the Go comment: `//go:flounder`. The directive should be placed before an expression which creates a channel. Such directive would allow the compiler to denote that the channel is used for communication with Flounder.

Next, the runtime of Go would be extended in a way that sending to such channel would block the sender goroutine. The goroutine would be placed into a run-queue after the sending has finished. In the case of error, the panic statement would be called in the context of goroutine.

The change would shift the semantics of send operation over such a channel, because the send would block the goroutine. However, to avoid being blocked from making a progress, we could spawn a new goroutine for sending.

5.1.2 Moving Event Dispatch to the Runtime

The current integration of the messaging system of Barrelfish relies on the Event Dispatch goroutine. The goroutine sends a message in the presence of an event to the related channel. However, we could argue that the event dispatching could be more tightly integrated into the runtime for sake of performance.

One possible improvement is to move the event dispatching to the run-time in the same way as the network poller of Go does. It would allow the run-time to schedule a goroutine which is waiting for a message immediately. Thus, the throughput of the channel integration would improve.

In addition, such an improvement would require to distinguish receiving from the integrated channels. It could be done with the compiler directive mentioned above.

5.1.3 Multicore Support

Adding multicore support would take an advantage of the parallelism. The parallel execution could be achieved because goroutines can be run on the Barrelfish threads which are scheduled on separate CPU cores.

In the current architecture of the integration, the multicore extension is relatively straightforward: we should make sure that sending and receiving over Flounder happens from the same dispatcher on which the related connection has been initialized. It could be done by pinning the Event Dispatch and the sender goroutines on particular Barrelfish thread with the `runtime.LockOSThread` function.

5.1.4 Preventing Stack Overflow

As it is mentioned in Section 3.1.2, the mechanism of calling external C functions does not use the contiguous stacks. Thus, such a call might overflow the stack. To avoid propagation of the overflow, we would suggest to apply any stack protection mechanism, e.g., the guard pages [49] technique could be used to catch the overflows.

5.1.5 Hake Integration

The source tree of the Barrelfish OS is built with the Hake build system [47]. Integrating the Go build process with Hake would allow developers to compile Go applications in the standard way.

5.2 Conclusions

In this thesis we have ported the Go programming language to Barrelfish. The port makes it possible to run Go applications on the OS. As a consequence, features of the language such as concurrency, type system and etc. can be leveraged when developing the OS.

During the porting we had to face several issues. First of all, the chosen Go compiler and run-time do not follow the same ABI conventions as Barrelfish

does. Next, the library OS architecture and the lack of POSIX signals does not map well to an OS model Go assumes.

As a part of the port, we have integrated Go channels with Flounder based messaging infrastructure of Barrelfish. The integration allows us to communicate with the existing Barrelfish applications over Go channels. However, because of the differences in semantics and implementation we had to consider some trade-offs such as loosing some type safety guarantees.

The channels integration makes it possible to write Barrelfish OS services in Go. As a proof of concept, we have implemented the nameservice replacement. Also, to easier integrate with the existing libraries ecosystem of Barrelfish, we ported the foreign function interface called CGO.

Finally, we have evaluated our port in terms of performance. First, we performed macrobenchmarks against the Go implementation of LevelDB and compared the results with the same benchmarks conducted on Linux. Despite the fact, that both operating systems are very different and Linux is more optimized, the database performs roughly the same on both system. Secondly, we compared the Barrelfish message passing over Go channels against the communication over the native Flounder interface. Because of the overhead we introduced, the integrated Go channels do not perform well. However, after applying some optimizations we managed to improve the performance. Thirdly, we have shown that the current behaviour of the run-time when executing blocking operations has a negative impact on the performance for short duration operations.

In spite of the performance drawbacks of the messaging system integration, our port brings the easy-to-use Go concurrency model to the Barrelfish OS. We argue that the model has some benefits when compared against the inherited event-driven model of Barrelfish. First, the concurrency can be achieved without the ”stack-ripping” technique. Second, it solves problems which occur when multiple threads rely on the same waitset. Third, the concurrency primitives can be used either within Barrelfish’s dispatcher boundaries or for inter-dispatcher communication. Finally, we have showed that implementation for a particular problem in Go requires less lines of code when compared to one written in C.

Appendix A

Channel Integration Artifacts

The source code of `flounder/example.go`:

```
1  package goping
2
3  import (
4      "flounder"
5      "runtime"
6      "unsafe"
7  )
8
9  //go:cgo_import_static go_flounder_goping_export
10 //go:cgo_import_static go_flounder_goping_connect
11 //go:cgo_import_static go_flounder_goping_send_ping
12 //go:cgo_import_static go_flounder_goping_send_pong
13 //go:linkname go_flounder_goping_export go_flounder_goping_export
14 //go:linkname go_flounder_goping_connect go_flounder_goping_connect
15 //go:linkname go_flounder_goping_send_ping go_flounder_goping_send_ping
16 //go:linkname go_flounder_goping_send_pong go_flounder_goping_send_pong
17 var (
18     go_flounder_goping_send_ping,
19     go_flounder_goping_send_pong,
20     go_flounder_goping_export,
21     go_flounder_goping_connect runtime.BfFunc
22 )
23
24 // Message types set by Flounder receive handlers
25 const (
26     msgIdPing     = 1
27     msgIdPong     = 2
28 )
29
30 // Channel for outgoing messages
31 type PingChanSend chan<- flounder.Serializer
32
33 // Channel for incoming messages
34 type PingChanRecv <-chan interface{}
35
```

```
36 type AcceptPair struct {
37     SendChan PingChanSend
38     RecvChan PingChanRecv
39 }
40
41 // Message types
42
43 type Ping struct {
44     Val uint8
45     Msg string
46 }
47
48 type Pong struct {
49     Val uint8
50     Msg string
51 }
52
53 // API
54
55 // The function opens connection to a service over Flounder.
56 func Connect(serviceName string) (PingChanSend, PingChanRecv) {
57     sendChan := make(chan flounder.Serializer)
58     recvChan := make(chan interface{}, 256)
59
60     bid := flounder.GenBindingId()
61     err := flounder.SetBinding(bid, recvChan, deserialize, nil)
62     if err != nil {
63         panic("Connect")
64     }
65
66     // Connect
67     sn := []byte(serviceName)
68     sn = append(sn, 0x0)
69     st :=
70         flounder.FlounderUserState(
71             runtime.Sysvicall3(
72                 runtime.BfFunc(unsafe.Pointer(&go_flounder_goping_connect)),
73                 uintptr(unsafe.Pointer(&sn[0])),
74                 flounder.Waitset,
75                 uintptr(bid)))
76
77     // Start sender routine
78     go flounder.Sender(st, sendChan)
79
80     return sendChan, recvChan
81 }
82
83 func Export(serviceName string) <-chan AcceptPair {
84     c := make(chan AcceptPair, 256)
85     bid := flounder.GenBindingId()
86     if err := flounder.SetBinding(bid, nil, nil, acceptCb(c)); err != nil {
87         panic("Export")
88     }
89
```

```
90      // Export
91      sn := []byte(serviceName)
92      sn = append(sn, 0x0)
93      err := runtime.Sysvicall3(
94          runtime.BfFunc(unsafe.Pointer(&go_flounder_goping_export)),
95          uintptr(unsafe.Pointer(&sn[0])),
96          uintptr(bid),
97          flounder.Waitset)
98      if err != 0 {
99          panic("Export#2")
100     }
101
102     return c
103 }
104
105 // Accept callback.
106 // Will be executed by EventDispatch routine when establishing
107 // a new connection with a client.
108 // BindingId corresponds to the new connection binding.
109 func acceptCb(c chan<- AcceptPair) func(
110     flounder.BindingId, flounder.FlounderUserState) {
111
112     return func(bid flounder.BindingId, st flounder.FlounderUserState) {
113         sendChan := make(chan flounder.Serializer)
114         recvChan := make(chan interface{}, 10)
115         err := flounder.SetBinding(bid, recvChan, deserialize, nil)
116         if err != nil {
117             panic("Connect")
118         }
119         go flounder.Sender(st, sendChan)
120         c <- AcceptPair{sendChan, recvChan}
121     }
122 }
123
124 // Serialization of outgoing messages
125
126 func (msg Ping) Serialize() (*runtime.BfFunc, uintptr) {
127     cStruct := flounder.Allocate(uint64(8 + 8))
128
129     // .Val
130     *(*uint8)(unsafe.Pointer(cStruct)) = msg.Val
131     // .Msg
132     msgPtr := flounder.Allocate(uint64(len(msg.Msg) + 1))
133     *(*uintptr)(unsafe.Pointer(cStruct + 8)) = msgPtr
134     flounder.GoStringToC(msgPtr, msg.Msg)
135
136     return &go_flounder_goping_send_ping, cStruct
137 }
138
139 func (msg Pong) Serialize() (*runtime.BfFunc, uintptr) {
140     cStruct := flounder.Allocate(uint64(8 + 8))
141
142     // .Val
143     *(*uint8)(unsafe.Pointer(cStruct)) = msg.Val
```

```
144     // .Msg
145     msgPtr := flounder.Allocate(uint64(len(msg.Msg) + 1))
146     *(*uintptr)(unsafe.Pointer(cStruct + 8)) = msgPtr
147     flounder.GoStringToC(msgPtr, msg.Msg)
148 
149     return &go_flounder_goping_send_pong, cStruct
150 }
151 
152 // Deserialization of incoming messages
153 
154 // The function deserializes incoming messages from Flounder.
155 // Each message is stored in an appropriate C struct and. A pointer to it is
156 // passed as cStruct parameter.
157 func deserialize(mid flounder.MsgId, cStruct uintptr) interface{} {
158     var r interface{}
159 
160     switch mid {
161     case msgIdPing:
162         r = deserializePing(cStruct)
163     case msgIdPong:
164         r = deserializePong(cStruct)
165     }
166 
167     // Free the struct
168     flounder.Free(cStruct)
169 
170     return r
171 }
172 
173 func deserializePing(cStruct uintptr) Ping {
174     val := *(*uint8)(unsafe.Pointer(uintptr(cStruct)))
175     msgPtr := unsafe.Pointer(uintptr(cStruct) + 8)
176     msg := flounder.CStringToGo(*(*unsafe.Pointer)(msgPtr))
177 
178     return Ping{val, msg}
179 }
180 
181 func deserializePong(cStruct uintptr) Pong {
182     val := *(*uint8)(unsafe.Pointer(uintptr(cStruct)))
183     msgPtr := unsafe.Pointer(uintptr(cStruct) + 8)
184     msg := flounder.CStringToGo(*(*unsafe.Pointer)(msgPtr))
185 
186     return Pong{val, msg}
187 }
```

A short version (retransmissions for the transient errors are excluded) of the source code of `go_flounder_example.c`:

```
1 #include "flounder_goping.h"
2 
3 // Export
4 // ------------------------------------------------------------------
5 
```

```
6  static void
7  export_cb(void *st, errval_t err, iref_t iref)
8  {
9      struct go_flounder_goping_state *gst;
10
11     if (err_is_fail(err)) {
12         USER_PANIC_ERR(err, "export failed");
13     }
14
15     gst = st;
16     err = nameservice_register(gst->service_name, iref);
17     if (err_is_fail(err)) {
18         USER_PANIC_ERR(err, "go_nameservice_register failed");
19     }
20 }
21
22 struct connect_event_params {
23     struct waitset_chanstate chan;
24     uint64_t binding_id;
25     uint64_t new_binding_id;
26     struct go_ping_binding *b;
27     void *st;
28 };
29
30 static void connect_event(void *p) {
31     struct connect_event_params *params;
32
33     params = p;
34     go_create_flounder_event(EVENT_TYPE_CONNECT,
35                              params->binding_id,
36                              params->new_binding_id,
37                              (uint64_t)params->st);
38     params->b->rx_vtbl = rx_vtbl;
39     waitset_chan_deregister(&params->chan);
40     free(p);
41 }
42
43 static errval_t
44 connect_cb(void *st, struct go_ping_binding *b)
45 {
46     errval_t err;
47     struct go_flounder_goping_state *gst;
48     struct go_flounder_goping_state *new_gst;
49     struct connect_event_params *params;
50
51     gst = (struct go_flounder_goping_state *)st;
52     new_gst = malloc(sizeof(struct go_flounder_goping_state));
53     new_gst->binding = b;
54     new_gst->binding_id = __sync_add_and_fetch(&go_flounder_binding_count, 1);
55     b->st = new_gst;
56
57     params = malloc(sizeof(struct connect_event_params));
58     params->binding_id = gst->binding_id;
59     params->new_binding_id = new_gst->binding_id;
```

```
60      params->st = new_gst;
61      params->b = b;
62
63      // Create and trigger chan for event on a binding waitset, because
64      // connect_cb is called on the default waitset and for flounder<->go we
            use
65      // the non-default one.
66      waitset_chanstate_init(&(params->chan), CHANTYPE_EVENT_QUEUE);
67      err = waitset_chan_trigger_closure(b->waitset, &(params->chan),
68                                         MKCLOSURE(connect_event, params));
69      assert(err_is_ok(err));
70
71      return SYS_ERR_OK;
72  }
73
74  errval_t
75  go_flounder_goping_export(char *service_name, uint64_t binding_id, struct
        waitset *ws)
76  {
77      errval_t err;
78      char *service_name_copy;
79      struct go_flounder_goping_state *st;
80
81      // We need to copy service_name, because it can be garbage collected.
82      service_name_copy = malloc(strlen(service_name)+1);
83      strcpy(service_name_copy, service_name);
84
85      st = malloc(sizeof(struct go_flounder_goping_state));
86      st->service_name = service_name_copy;
87      st->binding_id = binding_id;
88      err = go_ping_export(st, export_cb, connect_cb, ws,
            IDC_EXPORT_FLAGS_DEFAULT);
89
90      return err;
91  }
92
93  // Connect
94  // ------------------------------------------------------------------------
95
96  static void
97  bind_cb(void *st, errval_t err, struct go_ping_binding *b)
98  {
99      struct go_flounder_goping_state *gst;
100
101     gst = st;
102     gst->binding = b;
103     gst->ready = true;
104     b->st = st;
105     b->rx_vtbl = rx_vtbl;
106 }
107
108 struct go_flounder_goping_state *
109 go_flounder_goping_connect(const char *service_name, struct waitset *ws,
110                            uint64_t binding_id)
```

```
111 {
112     errval_t err;
113     iref_t iref = 0;
114     struct go_flounder_goping_state *st;
115 
116     err = nameservice_blocking_lookup(service_name, &iref);
117     if (err_is_fail(err)) {
118         USER_PANIC_ERR(err, "nameservice_blocking_lookup");
119     }
120 
121     st = malloc(sizeof(struct go_flounder_goping_state));
122     st->ready = false;
123     st->binding_id = binding_id;
124 
125     err = go_ping_bind(iref, bind_cb, st, ws, IDC_BIND_FLAGS_DEFAULT);
126     if (err_is_fail(err)) {
127         USER_PANIC_ERR(err, "bind failed");
128     }
129 
130     // Wait till connected
131     while (st->ready == false) ;
132 
133     return st;
134 }
135 
136 // Send
137 // ------------------------------------------------------------------------
138 
139 struct params_ping {
140     uint8_t id;
141     char *msg;
142 };
143 
144 struct params_pong {
145     uint8_t id;
146     char *msg;
147 };
148 
149 errval_t
150 go_flounder_goping_send_ping(void *st, struct params_ping *p)
151 {
152     errval_t err;
153     struct go_flounder_goping_state *gst;
154     struct go_ping_binding *b;
155 
156     gst = st;
157     b = gst->binding;
158     err = b->tx_vtbl.ping(b, NOP_CONT, p->id, p->msg);
159     if (err_is_fail(err)) {
160         USER_PANIC_ERR(err, "ping");
161     }
162     free(p);
163     return SYS_ERR_OK;
164 }
```

```
165
166 errval_t
167 go_flounder_goping_send_pong(void *st, struct params_pong *p)
168 {
169     errval_t err;
170     struct go_flounder_goping_state *gst;
171     struct go_ping_binding *b;
172
173     gst = st;
174     b = gst->binding;
175     err = b->tx_vtbl.pong(b, NOP_CONT, p->id, p->msg);
176     if (err_is_fail(err)) {
177         USER_PANIC_ERR(err, "pong");
178     }
179     free(p);
180     return SYS_ERR_OK;
181 }
182
183 // Receive
184 // ------------------------------------------------------------------------
185
186 static void rx_ping(struct go_ping_binding *b, uint8_t id, char *msg)
187 {
188     struct params_ping *p;
189
190     p = malloc(sizeof(struct params_ping));
191     p->id = id;
192     p->msg = msg;
193     go_create_flounder_event(EVENT_TYPE_RECEIVE,
194                              ((struct go_flounder_goping_state
                                     *)b->st)->binding_id,
195                              MSG_TYPE_PING,
196                              (uint64_t)p);
197 }
198
199 static void rx_pong(struct go_ping_binding *b, uint8_t id, char *msg)
200 {
201     struct params_pong *p;
202
203     p = malloc(sizeof(struct params_pong));
204     p->id = id;
205     p->msg_id = msg_id;
206     go_create_flounder_event(EVENT_TYPE_RECEIVE,
207                              ((struct go_flounder_goping_state
                                     *)b->st)->binding_id,
208                              MSG_TYPE_PONG,
209                              (uint64_t)p);
210 }
211
212 static struct go_ping_rx_vtbl rx_vtbl = {
213     .ping       = rx_ping,
214     .pong       = rx_pong,
215 }
```

Appendix B

Nameservice Implementation in Go

```
1  package main
2
3  import (
4      "flounder"
5      "flounder/gonameservice"
6      "fmt"
7      "regexp"
8      "sync"
9  )
10
11 // Hashmap for Name -> Iref mapping
12 var ns map[string]flounder.Iref
13 // List of clients waiting for a Name registration
14 var nsWaitingList map[string][]gonameservice.GoNameserviceChanSend
15 // RW-Lock to protect access to `ns` and `nsWaitingList`
16 var nsRWMutex sync.RWMutex
17 // Count for name classes
18 var nsClass map[string]int64
19 var classRe *regexp.Regexp
20
21 func main() {
22     ns = make(map[string]flounder.Iref)
23     nsClass = make(map[string]int64)
24     classRe = regexp.MustCompile(`([a-zA-Z]*)\.[0-9]*`)
25     nsWaitingList = make(map[string][]gonameservice.GoNameserviceChanSend)
26
27     // Export the service
28     c := gonameservice.ExportUnregistered()
29     for {
30         go handle(<-c)
31     }
32 }
33
34 // Request handler
35 func handle(p gonameservice.AcceptPair) {
36     for {
```

```
37          req := <-p.RecvChan
38          switch req.(type) {
39          case gonameservice.SetCall:
40              register(req.(gonameservice.SetCall), p)
41          case gonameservice.GetCall:
42              lookup(req.(gonameservice.GetCall), p)
43          case gonameservice.WaitForCall:
44              waitFor(req.(gonameservice.WaitForCall), p)
45          case gonameservice.GetCountCall:
46              getCount(req.(gonameservice.GetCountCall), p)
47          default:
48              panic("invalid_request")
49          }
50      }
51  }
52
53  // Register a new name
54  func register(r gonameservice.SetCall, p gonameservice.AcceptPair) {
55      nsRWMutex.Lock()
56      defer nsRWMutex.Unlock()
57
58      if _, ok := ns[r.Iface]; ok {
59          panic("register")
60      }
61      ns[r.Iface] = r.Iref
62      notifyWaiters(r.Iface, r.Iref)
63
64      if class, ok := class(r.Iface); ok {
65          if _, exist := nsClass[class]; !exist {
66              nsClass[class] = 0
67          }
68          nsClass[class]++
69      }
70
71      p.SendChan <- gonameservice.SetResponse{}
72  }
73
74  // Get Iref for a given name
75  func lookup(r gonameservice.GetCall, p gonameservice.AcceptPair) {
76      nsRWMutex.RLock()
77      defer nsRWMutex.RUnlock()
78
79      if iref, ok := ns[r.Iface]; ok {
80          p.SendChan <- gonameservice.GetResponse{iref}
81      } else {
82          panic("lookup")
83      }
84  }
85
86  // Wait for a name registration
87  func waitFor(r gonameservice.WaitForCall, p gonameservice.AcceptPair) {
88      nsRWMutex.Lock()
89      defer nsRWMutex.Unlock()
90
```

```
91      if iref, ok := ns[r.Iface]; ok {
92          p.SendChan <- gonameservice.WaitForResponse{iref}
93      } else {
94          if _, ok := nsWaitingList[r.Iface]; !ok {
95              nsWaitingList[r.Iface] =
                    make([]gonameservice.GoNameserviceChanSend, 0)
96          }
97          nsWaitingList[r.Iface] = append(nsWaitingList[r.Iface], p.SendChan)
98      }
99  }
100
101 // Get the number of names in a given class
102 func getCount(r gonameservice.GetCountCall, p gonameservice.AcceptPair) {
103     var retCount int64
104
105     nsRWMutex.RLock()
106     defer nsRWMutex.RUnlock()
107
108     if count, ok := nsClass[r.Class]; ok {
109         retCount = count
110     }
111     p.SendChan <- gonameservice.GetCountResponse{retCount}
112 }
113
114 // Helpers
115
116 func notifyWaiters(iface string, iref flounder.Iref) {
117     if w, ok := nsWaitingList[iface]; ok {
118         for _, c := range w {
119             c <- gonameservice.WaitForResponse{iref}
120         }
121     }
122     delete(nsWaitingList, iface)
123 }
124
125 func class(iface string) (string, bool) {
126     r := classRe.FindStringSubmatch(iface)
127     if len(r) == 2 {
128         return r[1], true
129     } else {
130         return "", false
131     }
132 }
```

Bibliography

[1] THREAD(2) Plan9 manpages. http://man.cat-v.org/plan_9/2/thread. [Online; accessed 16-September-2015].

[2] Ieee standard for information technology- portable operating system interface (posix) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages c1–3826, Dec 2008.

[3] cmd/cgo: do not let Go pointers end up in C. https://github.com/golang/go/issues/8310, 2014. [Online; accessed 16-September-2015].

[4] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[5] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of USENIX 2002 Annual Technical Conference*. USENIX, 2002.

[6] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[7] Joe Armstrong. The road we didn't go down. http://armstrongonsoftware.blogspot.ch/2008/05/road-we-didnt-go-down.html, 2008. [Online; accessed 11-September-2015].

[8] The Go Authors. https://golang.org/src/runtime/netpoll.go, 2015. [Online; accessed 16-September-2015].

[9] The Go Authors. Command cgo. https://golang.org/cmd/cgo, 2015. [Online; accessed 29-July-2015].

[10] The Go Authors. Command gc. https://golang.org/cmd/gc, 2015. [Online; accessed 28-July-2015].

[11] The Go Authors. Command ld. https://golang.org/cmd/ld, 2015. [Online; accessed 28-July-2015].

[12] The Go Authors. Effective Go. https://golang.org/doc/effective_go.html, 2015. [Online; accessed 11-July-2015].

[13] The Go Authors. Frequently Asked Questions. https://golang.org/doc/faq, 2015. [Online; accessed 11-July-2015].

[14] The Go Authors. The Go Programming Language Specification. https://golang.org/ref/spec, 2015. [Online; accessed 13-July-2015].

[15] Fransisco J. Ballesteros. The Clive Operating System. Technical report, October 2014.

[16] Team Barrelfish. Barrelfish Architecture Overview, Barrelfish Technical Note 000. Technical report, ETH Zurich, 2013.

[17] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[18] Dave Lance Cheney. Benchmarking Go 1.2rc5 vs gccgo. http://dave.cheney.net/2013/11/19/benchmarking-go-1-2rc5-vs-gccgo, 2013. [Online; accessed 07-September-2015].

[19] CoreOS, Inc. https://coreos.com/, 2015. [Online; accessed 08-September-2015].

[20] Ross Cox. Go, Open Source, Community. http://blog.golang.org/open-source, 2015. [Online; accessed 10-July-2015].

[21] Russ Cox. Bell Labs and CSP Threads. Technical report. [Online; accessed 16-September-2015].

[22] Russ Cox. Threads without Locks. https://swtch.com/~rsc/talks/threads07, 2007. [Online; accessed 06-October-2015].

[23] Russ Cox. Go Data Structures. http://research.swtch.com/godata, November 2009. [Online; accessed 11-July-2015].

[24] Russ Cox. Go 1.3 Linker Overhaul. https://goo.gl/8I4qm6, November 2013. [Online; accessed 28-July-2015].

[25] Russ Cox. Hacker News: Comment on "Golang is Thrash". https://news.ycombinator.com/item?id=8817990, 2014. [Online; accessed 11-July-2015].

[26] Prasun Dewan. COMP 242 Class Notes. Section 3: Interprocess Communication. http://www.cs.unc.edu/~dewan/242/s07/notes/ipc/ipc.html. [Online; accessed 6-October-2015].

[27] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M.Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[28] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[29] Facebook. RocksDb Overview. https://rocksdb.org. [Online; accessed 07-September-2015].

[30] Sanjay Ghemawat and Jeff Dean. LevelDB README. https://github.com/google/leveldb. [Online; accessed 09-September-2015].

[31] Goleveldb Authors. Goleveldb README. https://github.com/syndtr/goleveldb. [Online; accessed 07-September-2015].

[32] Ilya Grigorik. SSTable and Log Structured Storage: LevelDB. https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/, 2012. [Online; accessed 06-October-2015].

[33] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. Ac: Composable asynchronous io for native languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 903–920, New York, NY, USA, 2011. ACM.

[34] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[35] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[36] Richard Hudson. Go GC: Prioritizing low latency and simplicity. `https://blog.golang.org/go15gc`, 2015. [Online; accessed 07-September-2015].

[37] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[38] Baeten J.C.M. A brief history of process algebra. Technical report, Technische Universiteit Eindhoven, 2014.

[39] AT & T Bell Laboratories and M.D. McIlroy. *A research UNIX reader: annotated excerpts from the programmer's manual, 1971-1986*. Number no. 139 in A research UNIX reader: annotated excerpts from the programmer's manual, 1971-1986. AT&T Bell Laboratories, 1987.

[40] Gnu Libc Maintainers. The GNU C Library (glibc). `http://www.gnu.org/software/libc`, 2015. [Online; accessed 28-July-2015].

[41] Marlow, Simonz. Haskell 2010 Language Report. Technical report, July 2010.

[42] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[43] Peter Müller. Types and Subtyping. Concepts of Object-Oriented Programming. Lecture slides, ETH Zürich, 2014.

[44] Rob Pike. Go Concurrency Patterns. `http://talks.golang.org/2012/concurrency.slide`.

[45] Rob Pike. Go at Google: Language Design in the Service of Software Engineering. `https://talks.golang.org/2012/splash.article`, 2012. [Online; accessed 10-July-2015].

[46] Martynas Pumputis and Sebastian Wicki. A Task Parallel Run-Time System for the Barrelfish OS. Distributed systems lab report, ETH Zürich, 2014.

[47] Timothy Roscoe. Hake, Barrelfish Technical Note 003. Technical report, ETH Zurich, 2010.

[48] Scala Team. Scala Language Specification: Version 2.11. Technical report, 2015.

[49] Robert Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, 2005.

[50] Snappy Authors. Snappy README. `https://github.com/google/snappy`. [Online; accessed 07-September-2015].

[51] Manuel Stocker, Mark Nevill, and Simon Gerber. A Messaging Interface to Disks. Barrelfish Technical Note 015. Technical report, ETH Zurich.

[52] Symas Corp. Database Microbenchmarks. `http://symas.com/mdb/microbench/`, 2012. [Online; accessed 06-October-2015].

[53] Ian Lance Tailor. Split Stacks in GCC. `https://gcc.gnu.org/wiki/SplitStacks`, November 2011. [Online; accessed 11-July-2015].

[54] Ian Lance Taylor. The Go frontend for GCC. In *GCC Developers' Summit*, 2010.

[55] Ian Lance Taylor. Gccgo in GCC 4.7.1. `https://blog.golang.org/gccgo-in-gcc-471`, 2012. [Online; accessed 07-September-2015].

[56] LLGO Team. README. `http://llvm.org/svn/llvm-project/llgo/trunk/README.TXT`, 2015. [Online; accessed 28-July-2015].

[57] The Santa Cruz Operation and AT&T. System V Application Binary Interface. Technical report, 2013.

[58] Ken Thompson. Plan 9 C compilers. World-Wide Web document, 2000. Originally appeared, in a different form, in Proceedings of the Summer 1990 UKUUG Conference, pp. 41–51, London, 1990.

[59] Trustworthy Systems Team, NICTA. seL4 Reference Manual API version 1.2. Technical report, March 2013.

[60] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. *Sun Microsystems Laboratories*, 1994.

[61] Chothia Zaheer. Investigating OS/DB co-design with SharedDB and Barrelfish. Master's thesis, ETH Zürich, 2013.

[62] Gerd Zellweger, Adrian Schüpbach, and Timothy Roscoe. Unifying synchronization and events in a multicore os. In *Proceedings of the Third ACM SIGOPS Asia-Pacific Conference on Systems*, APSys '12, pages 16–16, Berkeley, CA, USA, 2012. USENIX Association.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Message Passing for Programming Languages and Operating Systems

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Pumputis	Martynas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 7. October 2015	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.