Using the Go Programming Language in Practice

Thesis · June 2014

2 authors:

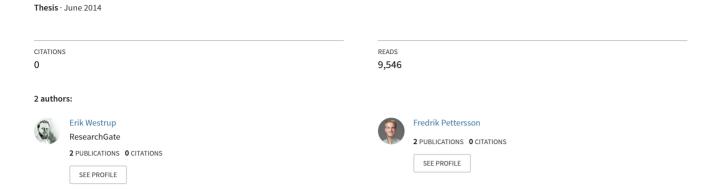# Using the Go Programming Language in Practice

Erik Westrup

<ada09ewe@student.lu.se>

<erik.westrup@gmail.com>

Fredrik Pettersson

<ada09fpe@student.lu.se>

<fredrik.pettersson.89@gmail.com>

June 9, 2014

Master's thesis work carried out at Axis Communications AB for the Department of Computer Science, Lund University.

Supervisors:
Jonas Skeppstedt <jonas.skeppstedt@cs.lth.se>
Mathias Bruce <mathias.bruce@axis.com>
Robert Rosengren <robert.rosengren@axis.com>

Examiner Jonas Skeppstedt

Abstract

When developing software today, we still use old tools and ideas. Maybe it is time to start from scratch and try tools and languages that are more in line with how we actually want to develop software.

The Go Programming Language was created at Google by a rather famous trio: Rob Pike, Ken Thompson and Robert Griesemer. Before introducing Go, the company suffered from their development process not scaling well due to slow builds, uncontrolled dependencies, hard to read code, poor documentation and so on. Go is set out to provide a solution for these issues.

The purpose of this master's thesis was to review the current state of the language. This is not only a study of the language itself but an investigation of the whole software development process using Go. The study was carried out from an embedded development perspective which includes an investigation of compilers and cross-compilation. We found that Go is exciting, fun to use and fulfills what is promised in many cases. However, we think the tools need some more time to mature.

Keywords: Go, golang, language review, cross-compilation, developer tools, embedded

Acknowledgements

Contents

Chapter 1

Introduction

1.1 Outline

Geeks, Computer Scientists and Engineers generally have always been looking for ways of simplifying their work, a practice that is very present among programmers. This fuels the steady and increasing stream of new programming languages being developed, some set out for solving a specific problem domain and some to be "the one language to rule them all". Here we have stumbled upon one of the new ones, *The Go Programming Language*[1], and we look into many aspects of it to find out what it brings to the table, what it tastes like and if we want more of it.

Go is a young language as it appeared publicly in 2009 and is backed up by Google engineers to solve the most common problems they experienced in software development, including long build times, hard to understand code and concurrent code organization among others. It is promised to be a clean and easy open source language with automatic memory management and built-in concurrency mechanisms. Furthermore it is also promised to be as easy to use as a dynamic language like Python as well as having the safety and speed of a statically typed language like C++ [1]. In other words, this is a language that is interesting to study more closely.

The promises of the designers of a language are not enough for making it a candidate for personal usage or for corporate adoption. A programming language is only as good as its compiler, development tools and community. At an early stage these are possibly flawed. A high level programming language is useless if it cannot be compiled down to executable instructions. We are also interested in knowing if Go is suitable to use for embedded programming, which is not an outset goal of the language designers.

Questions like these caught our interest and we have since programmed a lot in Go. We have tried many tools and reasoned about the language design compared to our previous knowledge in development with languages like C, C++, Java, Python, etc. Throughout

---

[1]From here on referred to as *Go*

the thesis we will sometimes refer to these languages as "mainstream languages". The comparison against C and C++ is very important since these are the languages that Go was designed to be a substitution for. They are also important from Axis' point of view since these languages are the main languages in Axis' development process. Through developing lower level software, cross-compiling and our eager of trying new things we have been able to build ourselves an opinion about Go and we share our findings and discussions in this thesis.

During this thesis we decided to take on a larger programming project in Go. We choose that project in such a way that it would expose us to the areas we were interested in. The project was to re-implement an existing service in an embedded product.

In chapter 1 we describe the background of this thesis and formulate questions we want to find the answers to. It is followed by chapter 2 where we describe in detail our research and findings. A discussion of the findings can be found in chapter 3. Finally the discussion leads to the conclusions in chapter 4.

1.2   Background

1.2.1   Introduction

In programming language discussions a quote from Lawrence Flon often shows up from his 1975 paper [2]:

> "There does not now, nor will there ever, exist a programming language in which it is least bit hard to write bad programs."

This is important to have in mind when trying a new programming language: we cannot expect it to be perfect because a bad programmer will always find a way of misusing the language structures. We should instead focus on how the language helps developers in using good coding practices. Go cannot let us do things that are not possible with other languages but the question is how the language lets us do it. For example thread synchronization can be achieved in Java but maybe it is easier to do in Go.

1.2.2   The Go Programming Language

Go has been described as "the C for the 21st century" and that it makes you feel like "being young again (but more productive!)" [3]. It is a compiled and statically typed language with C-like but simpler syntax and garbage collection. It was designed and developed to meet certain problems experienced by software engineers at Google. This company typically used Java, C++ and Python for their large projects and some of their projects they claimed to be indeed very large which makes some problems with the development cycle more evident [4]. Some of the major points were:

1. Build time not scaling well, which slows down the whole development cycle, partly caused by

2. limited code and package understandability in the team augmented by

3. the usage of different language subsets and patterns among developers (e.g. in C++) leading to

4. unnecessary safe guarding package imports that again augments confusion and slows down builds with languages like C where files typically has to be opened before reaching a header guard.

Go was designed specifically to address these points through:

1. Reduced build time with a model for dependency management.

2. Reduced bugs arising due to no pointer arithmetic[2] and by using a run-time garbage collector.

3. No implicit type conversions (that are often unexpected/forgotten),

4. Type inference i.e. no need to declare the type of a variable since its type can be inferred at compile time.

5. No complicated type system.

6. A concise language specification that has few keywords and lacks complicated constructs.

7. Language has built-in and easy to use mechanisms for concurrency.

One important thing to note here, which is also stressed by the creators of Go, is that concurrency is not parallelism. Concurrency is about organizing the code such that different parts can run at the same time along with synchronization and communication between the parts. Parallelism is about actually running things at the same time. This means that concurrency enables parallelism [5].

The built-in mechanism for concurrency is revolved around the so called *goroutine*. They function as lightweight threads that are handled by the Go runtime and it is cheap to start many of them (in scale of thousands). There is also a built-in type called *channel* that enables safe communication and synchronizations between goroutines.

The goal for Go was to fit in between the ease of use of dynamic languages like Python and the safety and speed of statically typed compiled languages like C++ or Java. The language was designed by the principles of being simple and clean i.e. the concepts should be easy to understand, and safety mistakes should be detected [6, p. 10].

1.2.3 Building & Compiling

The Go distribution ships with a set of compiler tools confusingly named[3] *gc* (Go Compiler) that targets the i386, amd64 and Arm platforms. Furthermore there is a front-end for *GCC* for Go programs, named *gccgo* [7]. This enables utilization of the GNU build chain that is familiar to many developers and the GCC's code optimization which is better than

---

[2]Not supported by normal Go pointers but can still be done if the type Pointer in the unsafe package is used.

[3]*gc* normally refers to a Garbage Collector which Go also has. In this thesis gc will refer to the compiler.

gc at the moment. The architecture used in many of Axis' modern products is *MIPS*, and also in this thesis since access to such devices were provided. At the time of writing, gc does not support this architecture but gccgo does (along with x86, x64, PowerPC, Alpha and more), so it becomes necessary for such use cases to build a cross-compiling GCC [8].

Besides the advantage of supporting more architectures and operating systems gccgo also supports dynamic linking of libraries while gc only supports the opposite, static linking [9, Why is my trivial program such a large binary?]. Linking libraries statically means that the external functions found in the library are resolved during compile time and copied into the target object file. Dynamic linking means that the functions are resolved and loaded into memory during runtime instead. The advantage of a statically linked executable is that it is self-contained. This means that the system does not have to be changed in order to run the program, which makes the program more portable. The disadvantage is that the executable becomes large in disk size. The advantage with a dynamically linked executable is that the disk usage becomes smaller as many programs can share the common code which is good on systems with scarce storage resources. The disadvantage is that the shared object has to be loaded into memory (takes time) and must fit in the RAM memory which could be a problem if the system has a small memory [10].

Later in the thesis we will compare the build tool in Go to other build systems like Makefiles, Autotools, etc. An important thing to have in mind here is that there will be no extensive survey about this, mostly because we consider this to be a master's thesis of its own. Instead we will provide a comparison from our own experience with these tools.

As we will talk quite a lot about *cross-compilation* in this thesis we need to establish what this means and the related terms in this context.

Platform A computer system setup consisting of a hardware architecture, operating system and its libraries [11].

Build platform The platform where the compiler is being built on i.e. where the compiler is compiled.

Host platform The platform where the compiler will run i.e. the system where the compiler is used.

Target platform The platform where the resulting executables compiled by the compiler will run [12].

Cross-compilation When the *host ≠ target*. In normal compilation they are the same.

Canadian Cross When *build ≠ host ≠ target* i.e. the compiler is built on one machine, runs on a second and produces executables for a third one.

1.3 Purpose & Goals

The purpose of this master's thesis is to investigate the challenges in introducing Go as the main language/platform for developing embedded platforms. We want to find out, in some sense, how good Go is in theory and practice as well as seeing if and how it solves problems programmers face in other languages. More specifically we want to know how

Go stands in an embedded programming environment which sets more requirements on the surrounding toolchains to be able to support multiple architectures. Companies like Axis are interested in knowing if adopting Go is affordable in the sense of education of employees and wants to know the answer to questions like if the language, tools, and community are mature enough for production. This means that the purpose of this thesis is not just to look at the language itself but the whole surrounding developer environment. This includes looking at the community, availability of resources, the future development of the language.

1.4 Problem Formulation

The questions in this section originated from the the initial thesis proposal from Axis. From the proposal we, with the help of our supervisors, could produce a more detailed list of interesting questions, which finally resulted in the questions below. Many of them are of subjective nature but throughout the thesis we try to give good arguments and discussions to support our opinions.

1.4.1 The Go Programming Language

The most important thing to study is obviously the language syntax, features and standard library.

- Is the language itself easy to use and understand when having a background similar to ours i.e. being adept in languages like C, C++ or Java?

- Is it easier and less error prone to write correct concurrent software in Go compared to other mainstream languages?

- Are the standard libraries and other common libraries mature enough for usage in production software development?

- Before learning and using a language there should be some type of assurance of the future and maintenance of the language, libraries, compilers and tools. Go is developed by Google, but how is the language governed now (and in the future)?

- How much can other parties influence the development of Go and what would happen if Google loses interest; is there a committee and community that could take over?

- Is a permissive license in use for the language specification and current implementation of tools and libraries?

1.4.2 Building & Compiling

No software is good if we cannot compile and use it; how well can we do this with Go programs:

- The Go build tools seem to impose a certain workspace directory structure; is it mandatory and limiting, hindering or complicated to integrate Go with existing build systems?

- Is it fast and easy to build a Go program?

- Why is it that static linking was chosen and is dynamic linking on the road map for gc?

- Building programs with the go tool looks easy for small programs, but will work well with real projects too? Can it be avoided to write explicit complicated build instructions e.g. Makefiles?

- Can a custom built gccgo be used with the go build chain as easy as when using gc, or does one have to resort to the classic GNU toolchain of configure scripts and Makefiles?

- How well does cross-compiling with gccgo work in practice?

- Will integration with C make the builds slower and more complicated?

- Can memory profiling be done in Go programs interacting with C? The fact that Go has a managed memory model with a garbage collector while C is manually managed makes for some possible complications.

- How does function pointer and callbacks[4] work between Go ⟷ C?

1.4.3 Development Tools

For Go to be adopted, there must be good accompanying tools to ease development. Considering the young age of Go, it might be that some tools are missing or not stable for production yet.

- How well established is the existing development tools for Go?

- Are there any tools that simplifies reading Go-code for example text editors that support Go syntax or code navigation tools like *Ctags* and *cscope* [13] [14]?

- Are there tools for debugging, code profiling, testing (automation and coverage analysis) or memory usage analysis? How useful are they?

- Are there any useful *IDE*s that support Go?

---

[4]When a piece of code, or pointer to such, is passed to a function that executes it at some later point.

1.4.4   Software Product & Development Qualities

What matters in the end is how the software product and development qualities are affected when using Go compared to industry established languages and tools. The most current standard of measuring software quality factors is *ISO/IEC 25010:2011*[5], categorizing qualities in 8 main categories: Functional Sustainability, Reliability, Performance efficiency, Operability, Security, Compatibility, Maintainability & Transferability [16]. Qualities are by nature subjective, so to make our questions measurable we will answer them from our experience with the language and tools. Finding more statistically satisfying answers would require a larger survey and study which is not in the scope of this master's thesis.

- How is development efficiency affected? Do the language and the tools help the programmer to focus on the important tasks?

- Is Go easy to adopt for programmers coming from C and higher level languages like Java or Python i.e. does Go offer good Understandability and Learnability?

- Is software written in Go maintainable, compared to other languages?

- How testable is Go-code? How good is the support for testing in the language standard library? Are there any useful third-party tools?

- In this thesis, practical testing of Go will be done for embedded devices which is a field that does not seem to have been the target for the language. How portable and reliable are Go programs?

- Are Go programs fast in terms of execution speed?

- Go has what is called a segmented stack (see section 2.1.3). Was the choice of having a segmented stack wise? Does it work well in practice?

1.5   Previous Research

When doing a language review like this, it would be good if there was a standard framework for doing so. Such a framework could specify which aspects of a language are interesting for a review. Then a comparison with other languages that have been reviewed with the same framework would be easy. However we have not found anything like this, which is not surprising as it would be a major project. On this subject we have found some course material that layout out a structure for programming language comparison aspects [17] [18]. However we found this outline obvious and we decided to not follow it, since we had already thought of the points raised there ourselves.

---

[5]Replaced the more known ISO/IEC 9126 standard [15].

1.6 Distribution of Work

This master's thesis was a joint project between the authors and the burden of work and times of joy were mostly equally distributed during the project. To make our work more effective, we divided some tasks between us during a few weeks. Fredrik focused more on the implementation of the project and testing while Erik focused more on cross-compilation tools and studies of the language.

Chapter 2

Approach

2.1 The Go Programming Language

The language was designed with simplicity in mind and therefore the language has few keywords, constructs and built-ins. But what is included is carefully put together and mixes well with the other language constructs. As a comparison, Go has 25 keywords whereas ANSI C has 32, Java has 50 and C++11 has 84 [6, p. 12] [19]. Even though the language specification is said to "be simple enough so that every programmer can have it in their heads", we are only going to focus on the features of Go that people might not be familiar with and what makes Go unique. For the ones that are interested in seeing more, the language specification is found on the homepage [20]. The recommended way of learning the language is to take the code-interactive guide "A Tour of Go" [21].

2.1.1 Syntax & types

The syntax will be familiar for programmers used to languages from the C-syntax family but it is clean and resembles how easy it is to write Python code. To start off, study the hello world program in listing 2.1.

Listing 2.1: Hello World

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello, World!")
}
```

First the package is specified, in this case the *main* package since this package will contain a main function. After that we have a list of imported packages, in this case the standard string formatting package. The imported package can either refer to a local package on the system or can be a URL as described in section 2.3.5. Then we see a declaration of the main function comes, with no arguments and no return value. As can be seen, statements in Go are not semicolon terminated[1]. Also notice that type visibility is determined by the first letter of the name like the *Println()* function from the *fmt* package. Type names beginning with a capital letter are exported whereas lower cased names are not visible outside the defining scope.

Go has the types a programmer would expect: integers, floating point numbers, strings, characters (UTF-8, named *rune*s and can be of variable size). Programmers new to the C-language often struggle with type declaration, this is because the declaration has to spelled out from the inside and follow a set of association rules to understand the type [22, 5.12 Complicated Declarations]. Go has a solution for this, variables are declared from left to right in the same order as one would read them as shown in listing 2.2. Line number 2 in the same listing illustrates how a variable's type can be deduced from the expression on the right of the special assignment symbol :=, which is known as *type inference*.

Listing 2.2: Types in Go

```
var i int // All Go types have sensible zero values, 0
    here.
j := 3.14 // Type is inferred, float64 on our machine.
str0 := "gopher"
var str1 *string = &str0 // str is a pointer to a string
```

Declaring types from left-to-right feels strange in the beginning for a C-programmer, but it does not take long before it becomes natural. To further illustrate how much of a difference this means for complex types, compare the two equivalent programs written in C and Go in listing 2.3 and 2.4. They declare (at line number 1) a new type that is an array of size one of functions that takes a pointer to integer and returns a string. It is worth to notice that in Go we do not take the address of a function as functions are "first-class" i.e. directly supported as a value.

Listing 2.3: Complicated type in Go

```
type funcCollection [1](func(*int) string)

var str string = "golang"

func f0(i *int) *string {
   return &str
}

func main() {
   fnColl := funcCollection{f0}
   input := 32
   fmt.Printf("%s\n", *fnColl[0](&input))
}
```

---

[1]The lexer, the initial parsing step during compilation, will insert them during compilation.

Listing 2.4: Complicated type in C

```c
typedef char *(*(func_collection[1]))(int *);

char *str = "clang";

char *f0(int *i)
{
  return str;
}

int main(int argc, const char *argv[])
{
  func_collection fn_coll;
  fn_coll[0] = f0;
  int input = 32;
  printf("%s\n", fn_coll[0](&input));
}
```

Another rather unusual feature of the language is that it supports multiple and named return values. Multiple return values are typically used for returning an error since exceptions are excluded from the language. If the return values are named, they are available in the scope of the function, reducing the number of variable declarations needed and it also makes it easier to trace which values are returned. When using named variables, no arguments have to be provided to the return statement as illustrated in listing 2.5.

Listing 2.5: Multiple and named return values

```go
func distance(p1, p2 uint) (dist uint, err error) {
  dist = 0
  err = nil
  if p1 > p2 {
    err = errors.New("the second point must be the farthest
        away from origin")
  }
  dist = p2 - p1
  return
}

func main() {
  var d uint
  var err error
  if d, err = distance(2, 7); err != nil {
    log.Fatalf("Invalid computation: %s", err)
  }
  log.Printf("%d\n", d)
}
```

There are several things to notice in that listing. In the argument specification for the function *distance()* we utilize that consecutive variables of the same type only need a type specification for the last value. We also see how error handling is done in Go at line 14 with the enhanced if-statement that includes an optional statement before the test expression. The call to the fatal logging function will print the error message and terminate the program with an error code.

Apart from simple arrays of compile-time known sizes there are dynamic arrays named *slices*. The composite type for Go is the *struct* which is the same concept as in C and C++. Furthermore Go has a built-in map type which functions under the usual key-value interface.

To make the language cleaner, parenthesis are not needed around conditions, return values are discarded by naming the variable "_", the *var++/var−−* incrementation/decrementation are demoted from expressions to statements so that there will be no confusion about which value is used. To handle clean-up work that is more advanced than what the garbage collector can do there is a *finally* concept in the language. A function can be put on a call-list, a list of functions that will be called when the current one finishes. It is easy to forget to do clean up in all of the possible return points of a function, so deferring the work to when and wherever the function return solves the problem. It is typically used like seen in code listing 2.6.

Listing 2.6: Deferred execution

```
func main() {
  f, _ := os.Open("/dev/null") // Discards error code.
  defer f.Close()
  // ...
} // f.Close() is called here.
```

2.1.2   Object-orientation

Since Go is a modern language we would expect it to include the popular concept of Object-orientation[2]. An object is often described as a structure that represents a concept, manages some information and provides means for operating on them. Is the C language object-oriented? Not built-in to the language, but Object-oriented behavior can be emulated by using a struct as the means for data store and then store function pointers in the struct being the operation that can be done on the object [23, p. 62]. It can even be taken further by implementing a dynamic dispatch table[3] for the operations and in that way support subclassing. Doing so would end up in a large library like *GObject* in *GLib* or even a language like C++ or Objective-C [24].

Go has structs in a very similar way to C but provides ways for operating on that structure which makes it OO. Methods on a struct can be specified by annotating a normal function declaration with what is called a *method receiver* in Go. A method receiver makes the specified type instance (can be of any type) available in the scope of the function and it also enables call on the struct's method with the classic dot notation. The example in listing 2.7 defines a struct named *Object* with a name field (not visible outside this package because of the first letter being small) and two method defined on it.

Listing 2.7: Method Receivers

```
type Object struct {
  name string
}

```

---

[2]Shortened to *OO* from here on, and OOP for Object-oriented programming.

[3]A way of selecting the correct method dynamically at runtime.

```go
func (o Object) GetName() string {
   return o.name
}

func (o *Object) SetName(name string) {
   o.name = name
}

func (o Object) String() string {
   return fmt.Sprintf("Object with name %s", o.name)
}

func main() {
   obj0 := Object{"obj0"}
   obj1 := Object{name: obj0.GetName()}
   obj0.SetName("tcejbO")
   fmt.Printf("%s\n", obj1)
}
```

In the main function we see two ways of initializing a struct: with positional or named values. Notice the difference between how the method receivers are declared in *Object.GetName()* and *Object.SetName()*. If the receiver is a type, it is a copy of the object and any changes to its field will not affect the original. To change the value we need a pointer receiver, which also avoids unnecessary copying. Method receivers make Go very extensible as we can define them for any type (in the same package though, which can be worked around with the soon explained concept of struct embedding).

The other key building block for OO in Go are interfaces which enables polymorphism [25]. Interface is simply a specification of method signatures under a name. They are abstract and thus do not qualify as a method receiver (it is not a concrete type!). We have already come across this in listing 2.7. Our Object struct implements the *Stringer* interface (in listing 2.8) from the standard formatting package [26] by having a *String()* method.

Listing 2.8: fmt.Stringer interface

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

Notice how we nowhere explicitly declared that Object should implement this interface[4]. This is how Go works; implicit interface implementations. Thus a (struct) type can implement multiple interfaces, namely all interfaces that consist of a subset of the methods that are defined for that type (determined at compile time). This resembles the ideas from *duck typing* (common in Python and Ruby for example) which is a type of programming where the provided methods for a type determines what it is – not what it declares itself to be [27]. This is another thing that makes Go feel dynamic while still being a statically typed language.

Furthermore all types implement the empty interface, which is Go's answer to void pointers in C or the Object type in Java i.e. the most general type. In Go, references to types come with information on the actual type so it is easy to convert the reference back

---

[4]As we would have done in e.g. Java with the *implements* keyword.

to the actual type using what is called a *type assertion*, possibly with a *type switch* that is similar to a switch statement [28, Interface conversions and type assertions]. A type assertion is illustrated in listing 2.9 where a reference of the empty interface is asserted to be a type that implements the quacker interface.

Listing 2.9: Type Assertions

```go
type quacker interface {
  quack()
}

func sound(animal interface{}) {
  duck := animal.(quacker)
  duck.quack()
}
```

Go is different in the way OOP is done compared to the mainstream languages in that there are no classes and there is no type hierarchy. The language is Object-oriented, but not type oriented. Go is designed around the design principle *Composition over Inheritance* which says using inheritance as a means for code reuse is bad since it creates unnatural relationships and breaks the open/closed principle when super classes have to be modified to embed a concept of a new subclass [29] [4, 15. Composition not inheritance]. It is hard to make the right design decisions in the early phases of a software project. This can lead to time-consuming redesign, code rewrite and factorization. As Russ Cox put it in a Google IO talk [30]:

> "The most important design decisions do not have to be made first, and it is easy to change types as the program develops…"

Go implements composition via what is called *struct embedding*. When an interface I embeds an interface J, all of J's declared functions are copied into I. When a struct A embeds a struct B, all of B's member variables will be included in the declaration of A and all methods defined on B will also work on A. An example of the syntax is shown in listing 2.10. This allows for easy pick-and-choose of functionalities, a concept that is known in some languages as *mixins* [31]. In Go programs, it is common to define small and coherent interfaces that can be reused by embedding.

Listing 2.10: Interface and Struct embedding

```go
type J interface {
  opJ() string
}

type I interface {
  opI() int
  J // J's definition is copied to I => I has both opI() &
      opJ()
}

type B struct {
  bData int
}

```

```go
14  func (b B) operate() {
15  }
16
17  // A contains aData & bData
18  // Can call operate() on A instances, but in operate() of
19  // course only the B part is accessible (think slicing in C
      ++)
20  type A struct {
21    aData int
22    B        // Unnamed type = struct emedding.
23  }
```

This is the (more flexible) replacement for subclassing and code reuse in Go. When a field in a struct is accessed, a name resolution is done starting at the outer most scope and continuing the search in the embedded structs if not found. This allows for method overriding [28, Embedding].

2.1.3   Goroutines & Channels

Goroutines are advertised as lightweight threads with a small initial stack, having little creation and context switch overhead. The Go runtime multiplexes one or more goroutines within native OS threads as well as moving them between threads. This is done to maximize resource utilization when blocking occurs in goroutines [28, Goroutines].

To make the goroutines cheap they start off with a very small stack segment of 8KiB (4KiB in earlier versions) which is grown on demand. The whole stack is segmented and growth and shrinking is accomplished by simply maintaining linked lists of segments. This makes it cheap to start a routine and easy to both grow and shrink as execution proceeds. However, this is now considered to be the wrong approach by the Go developers. Consider the amount of work done by creating and freeing segments if a function that allocated a lot of memory is called repeatedly. Because of this problem, referred to as "hot split" or "stack trashing", the next release of Go, 1.3, will have a contiguous stack allocation that grows and shrinks very much the same like a typical dynamic array [32] [33]. Other programming languages projects, like *Rust*, have come to the same conclusion after experimenting with split stacks [34].

Syntactically a new routine is started with the *go* statement followed by a function call. Often function call is to an anonymous function which is called a function literal in Go as shown in listing 2.11. These functions are closures meaning that they have access to the variable scope where it was created [20].

Listing 2.11: Starting a goroutine

```go
1  go work(data) // Call work() in a new goroutine
2
3  func process() {
4    data := getData()
5    go func() {
6      work(data) // Access parent's scope
7    }() // A call to the function literal
8  }
```

A Common mistake made by new Go programmers is how to use them. The function starting a goroutine does not wait for it to complete meaning that if code that depends on it is work has to communicate with it. When the main goroutine completes the program exits meaning that goroutines that have not completed execution or not even started execution will be stopped [20, Go statements].

To allow for inter-communication between goroutines *channels* are used. The idioms[5] that the channels enable are a bit different from what is common in other languages. For example in Java a common way of thread inter-communication is by using a data monitor. The Go-teams opinion about this is summarized into the Go concurrency slogan [28, Share by communicating]:

> "Do not communicate by sharing memory; instead, share memory by communicating."

Channels are used for exchange of data between goroutines and for state synchronization. A channel has one sending and one receiving end. Listing 2.12 shows that a channels has a type and a size which is specified when it is allocated and instantiated with the built-in make function.

Listing 2.12: Allocation and initialization of goroutines

```
1 syncChan := make(chan int)
2 asyncChan := make(chan int, 1)
3 asyncSemaChan := make(chan int, 8)
```

A channel always block on the receiving end until there is data to read. It is not "safe" to have multiple readers since reading the channel consumes the data exclusively. A synchronized channel (line 1 in listing 2.12) blocks on the sending end until there is a receiver on the other end. In other words, a synchronized channel synchronizes the execution state of goroutines.

An asynchronous channel (lines $2 - 3$ in listing 2.12) does not block on the sending end until the size is filled up (1 and 8 in the examples) so there can safely be multiple senders [28, Channels]. An asynchronous channel could easily be used for resource access control, a so called semaphore [35]. The syntax for sending and receiving is shown in listing 2.13.

Listing 2.13: Producer & Consumer with goroutines

```
1 func main() {
2   syncChan := make(chan int)
3   done := make(chan bool, 2)
4   go consumer(syncChan, done)
5   go producer(syncChan, done)
6   <-done
7   <-done
8 }
9 func consumer(input <-chan int, done chan bool) {
10   in := <-input // Wait & receive
11   fmt.Printf("%d\n", in)
12   done <- true
13 }
```

---

[5]The typical way a task is carried out in a programming language.

```go
func producer(output chan<- int, done chan bool) {
    output <- (1 << 3) // Wait & send
    done <- true
}
```

Notice here that the main goroutine waits for both of the other two goroutines to complete execution (but discards the actual value) and that the type for the communication channel between the producer and consumer is annotated with a direction making them read-only and write-only references respectively.

The initial plans for channels was that they should work over the network but that idea was soon abandoned since there are complications with synchronization over the network [36, 48:00]. In earlier versions of Go though, there was a package called *netchans* that wrapped channels over the network. But that package was removed since it was too complicated to use [37].

2.1.4   Standard library

The standard library for Go is comprehensive and has support for many things but it is not "fat". It does not have a myriad of different data structures like Java's Collection Framework nor does it offer conveniences like C++'s Standard Template Library algorithm module [38] [39]. As for the language syntax itself, every feature included is well thought of and is orthogonal (work well and naturally with) the other features to keep the language simple [6, p. 10].

There is a potential problem with the TLS implementation in the package *crypto/tls* however [40]. Andrew Gerrand states in the go-nuts mailing list that the package "hasn't been thoroughly reviewed from a cryptographic standpoint" yet [41]. This could be a problem for security critical applications.

2.1.5   Missing features

While Go has many of the normal programming language features as well as a couple of new ones, there are also concepts and constructs common in mainstream languages that are not present in Go. One example is that there is no *while loop*. Instead there is an enhanced for loop which can be used with only the conditional expression part. There is no *foreach loop* either, it is replaced with the keyword *range* that is used for iterating through arrays, slices, maps as well as channels. As mentioned before there are no exceptions, since error handling is done by returning error codes. Neither is it possible to do function overloading i.e. having multiple functions with the same name but different signature determined by arguments or possibly return value. The Go authors think that it could be nice to have, but in practice it makes for confusion and complicates the function dispatching. The same goes for operator overloading; it could be nice to have but it is too complicated to be worth it [9, Why does Go not support overloading of methods and operators?].

Generics

Go does not support generic programming like in Java[6] or C++'s templates. Generics let the programmer write algorithms operating on types that are not specified but satisfy some constraint e.g. being descendant of some type or implementing an interface. There are for example maps and channels in Go that can be of different types, but those are built-in to the language does not qualify under the name generics for the programmer. The Go authors are open to the question of including it in the language but currently they have not found a way of doing that without making the type system too complicated [9, Why does Go not have generic types?]. Currently it has to be done as it was done in Java before generics i.e. working with a type that all types are (Object in Java). In Go a reference to the empty *interface{}* would be used which then is unboxed using type assertion or with a type switch [28, Type switch]. While the language developers continue to discuss, the user community has worked around the issues. For example a Go library called *gen* that generates types from templates, similar to how C++ generates new types at compile-time for instantiated templates [43].

2.1.6   Legal

Go and the Go GCC front-end is released under the BSD 3-clause open source license with an additional right to use the patents that Google has, that are necessary for the provided implementation of the Go tools [44] [45] [46]. Further, Go is a true open source project meaning that the development takes place in the open and not behind curtains like some other Google projects like Android. Everyone can post to the Go development mailing list *golang-dev*, post issues to the bug tracker and submit patches [47] [48] [49].

Before a contribution can be accepted into the project, the author must sign a contribution license agreement [50]. This agreement is for the protection of the contributor and Google. It says that the contributor must only submit original work, that the work submitted can be distributed and derived, that the contributions are entitled e.g. the contributor is not employed under a contract that forbids contribution to open source projects.

2.2   Building & Compiling

2.2.1   The Environment

To get the Go tools and compilers on a system, one typically follows the official instructions and downloads a binary distribution or compiles it from source code [51]. Furthermore many GNU/Linux distributions provide binary packages for Go [52] [53]. The gccgo compiler is not shipped with the Go tool and has to be downloaded separately or compiled from source [54]. The Go tools work best if the recommended workspace setup is used. In Go a variable named *$GOPATH* is used and works like the POSIX *$PATH* variable in that it lists paths to Go workspaces [55] [56] in the file system. The Go tools then searches these paths when looking for packages which mean that Go build commands can be issued from any directory in a shell. This allows for organization of code into coherent workspaces

---

[6]Since JDK 5.0 [42]

while still being able to import packages between workspaces without the need to specify where those are since the tools searches for them in the $GOPATH. Each workspace has the following structure [57]:

bin compiled executable binaries.

pkg package object files i.e. compiled packages.

src source code divided into directories, one per package.

2.2.2 Building

When building a Go program there is usually no need for writing scripts or Makefiles to specify how it should be done as there is a whole plethora of build tools available [58]. In this thesis we have focused on the standard build-tool named and referred to as *go* because this is the most used tool. It is worth to mention that there are interesting build-tools targeting cross-compilation e.g. *goxc* but all of those that we have seen does cross-compilation with the gc compiler, which is not in our scope [59].

One of the problems the Go creators wanted to solve is how a C-compiler has to resolve imports by opening the same header files numerous times during a build and most often just discard it because of header guards (a flag that says it has been read before). To reduce time spent on resolving imports and opening files in Go, each compiled object file contains information about imported symbols so that other Go-files importing this symbols does not have to resolve the same dependencies again. Furthermore the exported symbols and data in an object file are placed early in the file so that reading can be done without having to read the whole file.

Working with the go build tool is easy. The first command in listing 2.14 builds and runs a Go program or package while the second shows how to just build a package.

Listing 2.14: Running a Go program

```
$ go run <*.go/package>
$ go build [-o output] [build flags] [<*.go/
  package>]
```

The compilation can be tweaked by passing arguments to the build tool such as choosing compiler or by passing flags to the linker. To learn more about what the tool does when it builds a package, run it with the *-x* flag to get list of actions taken. For the hello world program we would see that the go tool first creates a temporary work directory, then invokes the Go compiler and then the linker. In a more complex program with imports we would see compiler invocations on all imported files, build static libraries for each module and finally a linking of everything in to one executable.

2.2.3 gc & gccgo

Even though we sometimes refer to gc as one compiler it is actually a set of compilers based on the Plan 9 operating system[7] compiler toolchain [60]. This is not surprising since two

---

[7]The intended successor of UNIX, by Bell Labs.

of Go's designers, Ken Thompson and Rob Pike, also were involved in the development of Plan 9. In this toolchain architecture there is, for each target architecture, a separate compiler, linker and assembler. Each architecture has been assigned a random letter e.g. 5 for ARM and 8 for Intel 386. So for Intel 386 in gc the Go compiler, C compiler, linker and assembler are respectively named *8g, 8c, 8l, 8a* [61] [62]. These are the tools that the go build command invokes when gc is in use. Only the Go compilers had to be written from scratch, which is currently written in C but is likely to be translated/rewritten in Go soon [63].

When compiling with gc the default behavior is to produce a self-contained statically linked package containing all used libraries. If other C-libraries are referenced, the resulting binary is dynamically linked against *libc* and *libphtreads*[8]. Unfortunately gc cannot create shared objects, for that gccgo has to be used at the moment [64]. If gccgo is specified as the compiler, the resulting program is by default dynamically linked against all libraries including the go runtime functions in *libgo*. To get a self-contained program like gc produces, the flag *-gccgoflags '-static'* can be passed to the go build tool [65].

gccgo is developed separately from GCC and is manually merged with the GCC project from time to time. Although GCC is released under the GPLv2 gccgo is under the 3-clause BSD license [66] [46]. This means that the Go front-end code could be reused for other compilers for example the LLVM back-end which is under the BSD license too (meaning trouble to include GPL code) [67].

2.2.4 Cross-Compilation

It is relatively easy to cross-compile with the gc compiler, as described by one of the main Go developers, Dave Cheney [68]. When cross-compiling with gccgo there are two parts to it:

1. get a working GCC cross-toolchain and

2. use that toolchain with the Go build tool.

GCC

One could imagine that building a cross-GCC would be easy since the target platform for the resulting compiler can be specified when configuring its sources [69]. However the compiler is only one part in a working toolchain and will need an implementation of the C standard library (e.g. glibc, eglibc, newlib, uclibc, musl), helper tools for executables (binutils) and system headers (i.e. Linux header files) for any practical usage. Most of these tools are developed under different projects with different goals and release schedule. Also adding to this is that the tools are to be built on one platform and executed on another makes it even harder. Multiplying all different versions and combinations of a toolchain makes for a huge number of possible setups of a platform. This means that when a cross-compiling toolchain is built it is not unlikely it is the first time ever that this particular chain with those versions is built. This can lead to new problems that cannot be found even with Internet search engines.

---

[8]As can be seen by inspecting the binary with *readelf(1)* or *objdump(1)*.

The tricky part of building a toolchain is that the components are not separated and there are mutual dependencies which are not properly documented. This means that for instance the compiler and libc-implementation has to be partly built to bootstrap the other until one of them can be fully built. There are several tools that have been developed to solve this puzzle, for example *crosstool-NG* and Gentoo's *crossdev* [70] [71]. We successfully used crosstool-NG to build a working toolchain with GCC 4.8.1 but that version only supports an older version of Go. In the project we did in order to evaluate Go we needed to use language constructs that were included in newer versions. In other words, we needed to have GCC with version 4.8.2 or ideally from the 4.9 branch (which was not released at the time). Officially crosstool-NG supported version 4.8.1 but we were able to hack it to build version 4.8.2 by manually adding it to the install-wizard through a script among the files provided by crosstool-NG. The build failed but it still produced a seemingly working compiler. However, the actual performance of it was unknown and could hardly be trusted. Crossdev requires a Gentoo system which we did not have time to set up. For these reasons we ended up building the toolchain from scratch, which was a far more daunting task than we ever could have imagined. In the end we succeeded in finding the right recipe for our platform which we published as a script on GitHub with the hope to help others with similar problems [72].

The Go Tool

It was not obvious how to do cross-compilation correctly with the go tool using gccgo as we did not find any documentation on this. This led us to experiment ourselves and later start a discussion at the Go user mailing list *golang-nuts* [73]. From this we learned that the go tool is not easy to use directly in this situation. Cross-compiling tools have a "triplet prefix" in their name typically in the format of "<cpu>-<vendor>-<os>-<tool>" e.g. *mipsel-unknown-linux-gnu-gccgo* in our toolchain [74]. The problem with this is that go tool does allow for specification for such prefix so it will look for tools named "gc" or "gccgo". This can be overcome by making symbolic links and make sure that those are found before the local system versions in the executable search path [75]. Another problem we experienced is that the go tool produced command-line arguments to GCC and gccgo that are not supported (the *-m32/-m64* architecture size arguments) for a GCC compiler targeting MIPS. We first tried to fix the issue by submitting patches which started a discussion where one developer explained that the go tool must support the architectures that gccgo supports. To get such version of the Go tool, it must be compiled with gccgo itself (a gccgo targeting the developer machine that is) [73] [76] [77]. This process was however undocumented and currently does not work as we found a bug with the cgo tool [78]. As we were unable to find any documentation on cross-compilation with gccgo we contributed to the community driven Go wiki by creating a new page with a tutorial on the subject [79].

To get around these problems so we could compile our projects we ended up writing a wrapper script, *gomips* (see Appendix A.1), that sets up the environment with paths to the tools, sets the right linking flags and more. The scripts first run the go tool in dry mode[9] and records the compilation commands that would be executed. It removes the unwanted arguments that are not compatible with our cross-compiler and then evaluate

---

[9]When a program shows which actions it would normally have taken, like a preview of the real command.

the commands in a shell. This works as long as the Go tool reveals exactly all commands that would have been executed but is still a fragile solution as a new version of, or possibly some untried input to, the go tool could cause it to output commands that breaks our script.

2.2.5   C-Integration

Go is a general purpose language which is especially popular for writing web servers and related tasks. It is also popular as a system programming language which means that it will likely be picked up in environments where much software is written in C. It is therefore of importance that Go can integrate with C-programs if it is to be adopted in these environments. To solve this the Go distribution ships with a tool called *cgo* [80]. This tool lets the programmer import and access symbols from a C-library. C-code can be written in a comment in a Go source file. Such code and any imports will be available in a special namespace in the Go-code. The C/C++ compilers need directives such as include and linking flags which are also specified in specially annotated comments in the Go-code. Manually specifying include paths brings software compilation back to square one in some sense as one of the design goals for Go was to have easy builds. Fortunately these flags can also be generated for the C-libraries included by using the *pkg-config* support in cgo, which is a tool that is common in the C world [81].

During the build of a Go package that imports the pseudo package "C", the cgo tool reads the Go files and records compiler directives, extracts embedded code to separate C files and compiles them and any other C/C++ files found in the package directory. Go symbols can also be exported and used from C-code if they are preceded with a line containing a special export comment that cgo looks for, as shown in listing 2.15.

Listing 2.15: Exporting a Go function to C

```
1  //export gopher
2  func gopher() {}
```

During a build, C header files are generated for the exported Go functions and types and a header file describing the Go standard data types so that C programs can use the Go-code and types. Modules (Go or C) that are linked together must not only have the same target architecture but share the Application Binary Interface on how functions are called. The gc compiler has a different call convention from GCC meaning that C programs that should be linked with Go programs compiled with gc must also be compiled with the gc's C compiler. At the moment gc does not have a C++ compiler so in that case GCC and gccgo must be used [9, Do Go programs link with C/C++ programs?].

The cgo library further provides helper function for integration e.g. a function that converts a Go string to a null terminated C string. When programming with C care must be taken when it comes to memory allocation and pointers. Since Go is garbage collected such data should be put in a global declaration so that the garbage collector in Go does not pick it up, if the data is intend to be shared with C-code [82].

However we experienced a problem that the C-compiler and linker flags that we had put in our Go source files did not work and we had to specify them manually on the command line. The issue was that the build tool did not propagate the flags parsed out from the CGO-sections in our Go sources down to the C compiler tools. This was supposed to be done by letting the Go tool export the flags in an environment variable that could be

accessed by the other tools. It turned out that this variable was not being built correctly. First we submitted an issue to the Go project but after a while we got fed up by the bug and proposed patch that was accepted into the code base [83] [84] [85].

Other quirks includes that macros cannot be directly used in Go-code, but must be used in a wrapper C function [86]. While calling Go functions from C is supported, the main function must be in Go, to start the Go runtime, etc., which complicates situations when a library in Go should be used from a main C program.

Callbacks

C-libraries are often designed to let the user implement callback functions i.e. functions that called from the library on certain events. We were interested in knowing how this works when the callback function is a Go function. As mentioned Go functions can be exported and made available to C program [82, Calling Go functions from C]. The problem with this is that then the function must be called by exactly that name. When callbacks are used, we usually pass a function pointer to a function satisfying a specified interface. However, we cannot take the address of a exported Go function and pass that as the callback, since cgo will generate a name-mangles stub function in C-code uses a cgo function for calling between Go and C[10]. The user community wiki describes that a pointer to a Go function cannot be casted to the correct function type in Go code, but must be passed to a Go function that makes the cast. It illustrates in the same example how a callback from C to Go can be done. The problem with this example is that it assumes that the programmer can also write the C-code that makes the callback. This is not the case with for example proprietary third party libraries.

To solve this, we figured out that "gateway" functions must be used. For each Go function that should be called as a callback via function pointer from a C-library, create a gateway C function in the Go source file that does only one thing: call the corresponding Go function. Then the address to the gateway function is passed to the C-library. We experimented and set up a demonstration of how this works in appendix A.2. In this example, the Go program wants to be callbacked in a function *callOnMeGo()* but passes a pointer to a gateway function in C named *callOnMe_cgo* that simply calls callonMeGo. The C-library starts a new thread, which demonstrates Go relatively new capability of being called by threads not started by Go[11], and this thread calls the callback and waits for a result. One more quirk is demonstrated in this code, that Go files containing exported functions cannot co-exist with function C definitions. Therefore the *callOnMeGo_cgo* function has to be in another file. They can however co-exist if the C function is defined as static and inline. We contributed to the Go user wiki about cgo with some code example how to handle function pointer callbacks [82, Function pointer callbacks].

---

[10]Can be seen by using the *-work* flag to the go build tool, and then inspect the files generated by cgo

[11]Supported since Feb 2013 [87]

2.3 Development Tools

2.3.1 Testing

Go has a built-in test framework for automatic testing. Filenames with the suffix *_test.go* in a Go package are automatically compiled, along with the actual package, and executed by the *go test* tool. In these files all functions with the prefix *Test* or *Benchmark* are run sequentially in order with an instance of a testing struct implementing the interface for several test related functions like error reporting, test termination and test skipping. There is a built-in code coverage tool that reports annotated source code with coverage information. These tools in combination with the testing tool makes it easy to do the testing and to see what and how much was tested [88].

Besides the default testing package in Go there are a handful of good third party extension packages for testing. One example is the package called *testify* and is used to ease mocking and testing in general. It allows the user to easily specify function arguments to return value mappings for a mocked function [89].

Much effort was spent on trying different solution for function indirection that allows mocking out some functions of interest. The classic C-style approach of collecting function pointers in a struct could be used but this did not feel like the Go-way. We then decided to make interfaces for the functions we wanted to be able to mock. By doing this, switching out the real functions with mocked ones was trivial. Mocking a struct is easy by overriding a method with struct embedding. When testing, simply telling the library to use an instance of another struct that contains the mocking function is all that has to be done.

Another useful package is the package called *Ginkgo* that essentially is a Behavior-Driven Development testing framework [90]. This library allows for writing of easy to understand tests as well as giving the user the possibility to add setup and tear down before and after each test. There is also the possibility to add different contexts in which more specific setup and tear downs can be appended. Once Ginkgo is set up and some common scenarios are tested, it is easy to take a look at the code coverage report from the go tool and add a test that covers a new untested function. In the background of Ginkgo the library *Gomega* which is a matcher/assertion library [91]. Ginkgo can also be compiled into a binary for running tests instead of using the normal test tool in Go. It adds special output for ginkgo-tests as well as running other tests just as normal.

2.3.2 Debugging

Searching for a bug in a computer program is usually done by printing output at relevant sections in the code (or possibly by using assert statements, which Go does not support) or by using a debugging tool. A debugger lets the user stop the program execution and inspect the state of register, variables and arbitrary memory sections. Advanced debuggers will additionally map the machine instructions back to source code and let the user step through the code line by line. To enable these features the executable must be annotated with extra debugging information. The Go compilers insert debugging symbols to the executables by default using the *DWARFv3* format [92] [93].

The most widely known debugger is probably *GDB*[12] which is an advanced command-line debugger supporting programs written in many languages including C, C++, Objective-C and Go [94]. GDB automatically looks for a section in the executable file named *debug_gdb_script*, which can contain a path to a runtime extension to load. Go currently ships with a GDB-extension which is loaded in this way. This extension makes packages, functions, Go types and goroutines available for inspection with GDB commands.

During the course of our work we often updated the compiler and tools to the latest versions and after one update the compiler stopped writing the debug script section to the executables. After filing a bug report we learned that the Go developers are not satisfied with the current solution of extending GDB [95]. One of the problems with GDB is that it follows OS threads whereas, but a programmer debugging a Go program wants to follow goroutines. Furthermore, at the time of writing[13], programs compiled with gccgo is not supported by the GDB runtime script. Therefore the developers removed the functionality. However, about a month later it was added back. But during the time it was disabled something else must have changed as the runtime script is currently not working which led us to submit another bug report [96].

The problem with GDB and goroutines is actually a problem we experience ourselves when we wanted to inspect the state of the goroutines in a core dump[14] from a crashed program by loading the core dump into GDB. It turns out that the GDB Go runtime extension cannot read the goroutine states from a core dump, which is problematic for embedded programs where a core dump is the only thing available for inspection after a crashed program [97]. One of the original Go authors, Rob Pike, has said that full GDB support is not a priority and will probably never happen [98]. The first option in debugging i.e. using print statements is still easy to do in Go because of the fast compile times.

2.3.3 Documentation

In Go there is a built-in tool that produces documentation for Go-code called *godoc* [99]. When a package is passed to godoc it will parse the source code along with the comments within the package and output the documentation on standard output. It is also possible to provide more arguments to the tool so that it produces an HTML page for the documentation. The resulting web page for the documentation looks the same like the documentation for the Go standard library. Other arguments can also be specified to the tool for setting up a server so that the documentation can be reached by other people in the network.

In order to add documentation for something, all that is needed is a preceding comment with no intervening blank lines. There is no special syntax that has to be followed in the comments; instead it is up to the developer to provide the documentation best suited for the situation. The convention however, is to begin the first sentence with the name of the element it describes.

If a package is hosted on a website e.g. GitHub, there is a service available at godoc.org that generates and displays the documentation for them. Getting documentation for a package is then as simple as entering the path to it in the address field. The documentation for the Ginkgo project for example, can be generated and viewed by visiting the

---

[12]The GNU Project Debugger
[13]2014-05-07, gcc version 4.9.0 20140307
[14]A copy of the process' memory at the point of crash.

URL godoc.org/github.com/onsi/ginkgo. The tool supports all "go-gettable" paths/packages (described more thoroughly in Section 2.3.5).

2.3.4 IDEs & Text Editors

One of the oldest problems in history of collaborative software development is that each programmer has its own preference in coding style for example on how to do indentation and where to place braces, which is very chaotic in the C world. Now, most modern languages have a defined coding style standard that is recommended to follow. Go has such coding styles as well but takes it a step further and ships a formatting tool called *gofmt*. This tool parses the input Go source files using the built in parser packages for Go, builds up a syntax tree and the outputs the code in a correctly formatted fashion. This tool can easily be added to hooks in code version control systems or integrated to text editors and IDEs. Having a tool that formats code to a set of rules is not a new concept, for example the old Berkeley tool indent and its GNU reimplementation [100] has existed for a long time. But what is new here is that the tool is shipped with the Go distribution and has one, and only one way of formatting the code: according to the Go coding style. Furthermore, gofmt has refactorization capabilities that are called "rewrite rules". This rules will rewrite matching Go expressions to some replacement expressions [101].

With the Go installation comes various tools (the main feature being syntax highlighting) for different text editors and IDEs, for example *Vim*, *Emacs*, *IntelliJ IDEA*, etc. During the course of this thesis we have mainly used two different text editors, Vim and *Sublime Text*.

There are more development tools than what is offered by the Go project for Vim, the most popular one being *vim-gocode*. This plug-in adds code completion, auto-formatting with gofmt and other IDE-like features that are nice to have [102]. There is a similar plug-in for Sublime-Text called *GoSublime*.

To easily locate definition in source code, the tool Ctags can be used and it has support for Go. Another small tool called cscope, that is used for browsing code can also be used for Go-code. Although their home page states that cscope only support C, C++ and Java, we have empirical proof of that it works for Go as well.

Several IDEs have support for Go but the only IDE that is specifically developed for Go is *LiteIDE*. The IDEs that have the best support, through plug-ins, for developing Go-code are IntelliJ and Eclipse.

2.3.5 Other

Linter

The Go standard library contains an actual Go source file parser in the *go/parser* package that outputs an Abstract Syntax Tree representation of the program [103]. This enables the development of tools that can analyze Go programs. An example of such a program is a linter which is a tool that inspects a source file for possible mistakes [104]. We have tried the tool *golint* that parses input files with the Go parser and gives the user handy tips on how to improve the code [105].

Package Manager

Go comes with a built-in tool for package managing called *go get* [106]. When given the location to a package (a URL), *get* downloads and installs it as well as all its dependencies. The path that is provided by the user needs to follow a special syntax depending on what code hosting site it is. There is a default support for Bitbucket, GitHub, Google Code and Launchpad but with a little bit of configuration it also works for code hosted on other servers. The tool checks for a *<meta>* tag in the HTML to discover where the code resides. The version control systems that are supported are: Bazaar, Git, Mercurial and Subversion.

2.4 Physical Access Control System

The many aspects of this thesis could be studied and tested separately. However, we was considered to be more efficient and interesting to develop a larger project and exercise the points of interest. A larger project is more likely to reveal the true nature of Go-programming and expose more aspects of the language, both advantages and disadvantages. We choose the project such that we would naturally come in contact with low-level programming, cross-compiling, Linux-programming, Interprocess communication and integration with larger C-libraries. The software we developed was a slimmed down reimplementation of an existing system written in C. This opens up for comparison with existing software. Here follows a short description about the system.

This thesis was carried out at a Swedish company, Axis Communications AB, based in Lund that is widely known in the network video camera business. The company recently entered the market of access control by releasing a network attached door control system: "AXIS A1001 Network Door Controller" [107]. These devices are known as Physical Access Control Systems or *PACS*. It is the first non-proprietary and open IP-based access controller on the market. The idea behind the product is that it should be easy to install and maintain while still being configurable and in this way support more advanced solutions [108]. This is done by supporting many different card readers, switches and door locks as well as providing an open API [109]. The device is connected to and powered over Ethernet with *PoE*[15] and has a number of input and output pins where devices can be connected. The software in PACS is written in C and the architecture is built up by several daemons using an *IPC*[16] system for communication.

A typical setup of a PACS is as follows, and depicted in figure 2.1. A *RFID*[17] card reader with a key set is mounted to the wall and connected to the PACS unit. A user swipes his or her card and enters an associated PIN code. The daemon handling card reader events receives the input from the card reader. The information collected by the card reader is then sent to the decision daemon which either denies or grants access based on the configured security scheme. If access is granted, the daemon controlling the door is notified which opens the connected lock for a preconfigured amount of time.

This is the basic structure of PACS but there is of course more to it. For example the web-interface for configuration, the API, the database and so on. The plan for this thesis

---

[15]Power over Ethernet
[16]Interprocess Communication
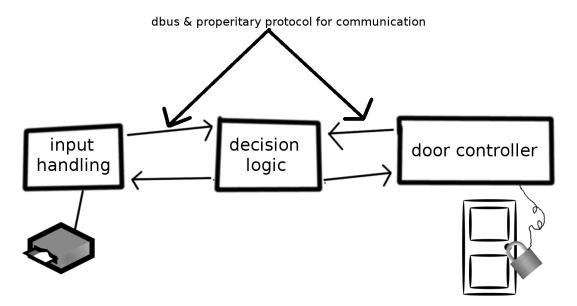[17]Radio-frequency Identification

Figure 2.1: Structure of the Go-implementation of PACS.

was originally to reimplement most parts of PACS in Go but because of the small time-frame of a master's thesis, we considered it to be a good idea to just focus on the three core daemons. Even this was a huge project to take on so the final Go implementation is a simplified version of these three daemons. For example the decision daemons database is just an in-memory map where the card-numbers with their respective PIN-codes are stored. Furthermore, the communication between the daemons is just a simple *D-Bus*[18] solution. However, the parts we did implement included low level bit flipping, module design, use of many Go features, concurrency and IPC. Because of the fact that we did a larger project, things like testing, debugging, documenting, etc. came naturally in to this project.

The plan was to use the same library for IPC just like the real PACS, instead of just using plain D-Bus calls. Due to a needed platform update for PACS and there not being enough time it was not possible to achieve this goal. However, we managed to make a proof of concept on an Axis camera (with the update) and because of this we got first-hand experience in how to use large C-libraries in Go. The library uses many *GLib*[19] features and therefore we also got experience in working with the library in Go.

---

[18]A message bus system, a way for applications in user space to talk to each another [110].

[19]A C-library that provides core application building blocks [111].

Chapter 3

Discussion

3.1 The Go Programming Language

The structure of the discussion will mostly follow that of chapter 2. To start of the discussion of the covered material we discuss the programming language itself.

3.1.1 Less is More

In a blog post Rob Pike, one of the Go authors, talks about the story behind the creation of Go as well as explaining Go's simplifications over C and C++ [112]. Before working with the Go-project, Pike was a C++ programmer and for various reasons he did not like the language. The idea was to create a language better suited for solving the problems where he currently used C++. It was also meant to attract other C++ programmers and make them see Go as an alternative. But now, a couple of years after the release, very few of the existing Go-programmers come from C++, Pike says. Instead, most programmers come from languages like Python and Ruby. The reason for this trend can probably be narrowed down to the mindset of many C++ programmers which is: the more features a language has and the faster the programs execute, the better. The idea behind Go is the complete opposite, minimizing programmer effort is much more important. As Pike puts it [112]:

> "Less is exponentially more."

When we started this thesis we had very little knowledge of Go and close to no experience in developing Go-code. One of the best thing about Go is how easy it is to learn. We started with going through the interactive "Go tour" at the Go website [21]. It took a few hours to complete, which should be the case for someone who already know programming in an imperative language. The tour explains most of the unique concepts and structures of the language. After that, it took us around a week of using the language to really be productive and we believe that this is the normal case as well. From our own experiences,

we feel that Go has the edge over any other language in the sense of how easy it is to learn for people who know how to program in another mainstream language. Since the language was designed with the philosophy of simplicity and orthogonality, the standard library is not complex and learning how to use it is also easy. It has some impact though, an example is that Go only provides a small set of data structures, unlike for example Java's Collection Framework. This means that when a specialized data structure is needed, e.g. hash map with predictable iteration order like the *LinkedHashMap* in Java, the Go-way is to manually implement it. Because of the fact that the community behind Go are developers that come from different programming languages with different mindsets, it is likely that the data structure has already been implemented by someone else. The initial thought about simplicity and orthogonality is very important for the Go-team and for the purpose that the language was designed to solve, the data structures that exists are proved to be enough. This probably means that we will not see an increase in the amount of data structures available in the standard library, at least in the near future.

Go was designed to solve specific problems that Google has, as described in section 1.2.2 which includes the goal of making the whole development process smoother and faster. To achieve this, the focus has been on the language construction itself and so far not on making the compilers optimized, as that is something that can be done later. Go is not a slow programming language but it is not the fastest one either. There is a project that compares the execution time, memory usage and the lines of code needed for programs written in different languages. From this site we can see that Go is, on average, about three times slower than C and C++ but needs few lines of code in return [113]. This is something that the Go-team is currently working on and they hope that Go will be one of the top contenders when it comes to execution time in the future. More specifically, they are aiming to be generally faster than C++ after the release of Go 1.3, that will be released in June 2014 [114, 12:08].

When dealing with a young language, it is good to prepare for stumbling upon things that are not quite as they should be. Go is not an exception from this and during our work we have found a couple of bugs in the standard library. For example we had trouble with incompatible constants and system functions working incorrectly. This is where Go being open source comes in handy, the only thing that has to be done is to report the issue on the Go-project homepage and if a fast fix is desirable (if the source of the problem is known), submit a solution to the problem [115].

3.1.2 Syntax & Types

There are several constructs in Go that, after using them for a while, we realized they do not only make it easier for the developer but they also make the code concise and more understandable. One design decision of the language that makes it easier to learn is that the types are declared from left-to-right. Having multiple and named return values is something that many other languages do not have support for and a topic that many have different opinions on. An example of one approach to support multiple return values is the keyword *out* in C#. Although this works, we feel that the Go approach is cleaner and many languages would benefit from having it. Naming the return values reduces the number of variable declaration while documenting the function's interface at the same time, which is another thing that most programming languages do not support.

We think that embedding type visibility in the name itself, determined by the case of the first letter, is a really smart as the visibility is known by just seeing the name. A possible problem with this approach could be if there would be a third or visibility mode added to the language at a later point. Then using use the case of the first letter is not enough to represent the visibility. As long as there are only two types of visibilities though, this works very well.

Even though it possible, but not recommended, to import a package to the same namespace[1] we like that this is not common to do. It is really helpful when the full library name is used with external names because then it is clear where that name comes from. It is a little longer to type, but the readability and understandability benefits are much greater.

In Go there are only two keywords that has to be known in order perform all kinds of iteration: *for* and *range*. The fact that the Go-team has made the decision to remove all extra constructs for iteration is something that adds simplicity, especially for people new to programming. To make the common pattern for testing if a key exists in a map and then fetching the corresponding value simpler, Go has these two steps combined into one operation. Go includes other smart ways of making development simpler e.g. uninitialized types are given a predefined zero value, built-in dynamic arrays, etc.

There are many other afterthought constructs that are aimed to be concise while still conserving the understandability. Go is a statically and strong typed language but has type inference that makes it easy to work with variables. The otherwise stuttering variable declarations are now simple and short[2]. We find this useful and comprehensive but there are situations where, if not used carefully, $:=$ can produce bugs which are hard to spot. As described in section 2.1.1 Go's if-statement takes an optional statement before the expression. Variables declared here are available in the if-block's scope. This can easily leads to programming mistakes. When we intended to assign a value to a previously declared variable, we used the type inference assignment $:=$ which will make a new variable shadowing the other variable. The effect is that the values assigned to the new variable disappears with that scope and are not available after, as intended.

While the possibility to defer function calls solved the problem of clean-up work they can be annoying to use. During the implementation of the PACS project we were debugging a module and had forgot that it contained a deferred call to a clean up function. Thus we were very confused when we could not explain why the execution did not follow the path we thought it would do. Experience Go developer will have this in mind when debugging code and code should be divided to small units so it becomes harder to miss defer statements.

3.1.3 Object-orientation

A good thing about Go is that it makes it easy for the programmer to follow good practice. In the mid-90s, people were really excited about code-reuse through inheritance. Many of the most used mainstream languages evolved during this time and they were of course influenced with this new way of programming. It did not take long until it was discovered that inheritance can easily be abused and gives severe consequences on the effectiveness of development. As early as 1995 in the famous "Gang of Four" book on programming

---

[1] By prefixing the import by a single dot.
[2] *message := "Hello"* instead of *var message string = "Hello"*.

design patterns it was concluded that one should "favor object composition over class inheritance" [116, p. 32]. Go was designed long over a decade since that and puts emphasis on practicing code reuse by composition. After having played around with composition in Go and thought about program design, we have also come to the understanding that inheritance can be clumsy. Go makes composition easy to use with struct embedding.

While it is flexible to let a type's capabilities be defined by what it can do and not what it is, we also see some problems and possible abuses with it. One of the pitfalls is to include too much functionality for a type. This will, without proper documentation, make it hard for a programmer new to the module to grasp the scope and design of the types. It makes sense to combine file write and read functionality to a single type but adding for example graphics rendering functions to the same type makes the type non-coherent and bloated. Too many levels of struct embedding can also be confusing. The freedom given by composition should be used to advance the program development and code reuse but not be abused to create a vague program design. The same things goes with fact that interfaces in Go are implicitly satisfied. While we like the flexibility it can be hard to spot what a type is. That is on the other side not a relevant question for a Go program, since a type's capabilities defies it.

3.1.4 Generics

When we first started to read and learn about Go and heard that there was no generics included we thought this was a major shortage in the language. A Java or C++ developer will ask "how can I write general code without assuming anything about the types if there is no generics?". As described in section 2.1.5 it is not hard to get the same functionality, it is a bit more cumbersome though. Take for example a linked list containing an unspecified type, which is easy to implement and use in Java. Is the linked list in the Go standard library equally easy? Implementing it, instantiating and putting values into an instance works without trouble since the empty interface that all types satisfies can be used. The trouble comes when we want to get the value contained in a list element. Then we will have make a type assertion [117].

We have to ask ourselves, how bad is it that we have to do some type assertions once in a while? Certainly it would be cleaner and less cumbersome if we could get the actual type directly. However, Go is not type oriented but more of a duck-typing language. This means that one should not have to know the actual type, but what it can do. Even in those cases when we want to get the actual type back, a type assertion is a simple expression. The Go authors say that adding generics would complicate the language. If that is true, we pay a very small price for keeping the language simple; unboxing a type is not really that awful. The Go developers state that they are open to adding generics if it can be done in a good way orthogonal to the existing features. But considering that they still have not added it, it is not likely to be added soon.

3.1.5 Concurrency

We found it easy to work with Go's concurrency model, using many goroutines, passing messages and synchronizing with channels. We also found, after coding a couple of weeks in Go, that we used goroutines and channels a lot, a lot more than we would have

used threads and mutexes/synchronizations constructs in other languages. In other languages, we often feel that bringing in a thread is a last resort since it complicates the code. In Go however it feels very intuitive to separate work to goroutines and communicate with channels. The amount of work to do is minimal in Go since this is built into the language. Compare with using e.g. POSIX threads for signaling: both a mutex and a condition structure must be initialized, take the mutex and call a wait function [118]. This procedure can be learned by heart and is doable, but all these steps and different arguments to the initialization functions create many possibilities for programming mistakes. Go eliminates this erroneous work.

Go's channels works well for intraprocess communication between threads but if interprocess communication is desired for example UNIX Sockets (and libraries on top of that like D-Bus) has to be used or TCP/UDP sockets for network communications. The problem of seamless intra/inter-process and network communication and concurrency is not solved by Go channels but there are many projects aiming for that e.g. ØMQ (has bindings for Go too) [119].

We can also compare with Java which has a built-in concurrency mechanism for thread-exclusive access to objects methods. Those methods are declared with the *synchronized* keyword to solve communication by modifying an object's state [120]. This opens up for many mistakes since the objects data could be modified from a method on that object which is not synchronized (as a mistake) meaning that the programmer must have discipline to synchronize the correct methods and keep track on where an object's variables are used. Java provides many ways of dealing with concurrency e.g. solving the producer-consumer problem with the *java.util.concurrent.BlockingQueue* interface. But the problem is just that, that there are many ways of doing it and no standard way that works well in all cases so the programmer has to learn many APIs and read documentation on how to achieve concurrency instead of practicing it. goroutines however solves message passing and synchronization in an easy and general way.

One thing we learned the hard way is that Go makes it clean and easy to write concurrent software but it does not solve the problem of writing correct concurrent software i.e. the problem of concurrent software is as hard as always but hopefully it becomes harder to do the mistakes when it is easy to see what is written. Go ships with a tool for detecting "racy behavior" at runtime but Go developers still have to keep their design clean and understandable to be able to write correct software [121].

3.2 Building & Compiling

3.2.1 Building

The build tool is one of the best features with Go according to us. We feel that too much time is spent on configuring how to build programs. The situation for C projects is often to write Makefiles, using CMake or Autotools with magic rules that compile packages and their dependencies [122]. These build files easily become hard to understand, resulting in that build configurations are made once and then copied to other projects without adjustments, resulting in slow or even erroneous builds. There are build tools like Apache Ant and Maven for Java that are simpler to use, since the build files are written in XML. But it

still requires the developer to read a tutorial or manual and can suffer from the same problems as the C build tools. Having a tool like Go's build tool is invaluable when a new code project can be started with zero cost since there is no need for setting up the build environment. All that is required is to create another directory in one of the Go workspaces that is included in the $GOPATH. With a tool that figures out how a package must be built and which dependences must be compiled the developer does not have to know exactly what is going on during a build. For most developers this is not a problem but a blessing. A possible negative effect of this is that the developers has less clue about the steps involved in building and which relations exists, which could be necessary in debugging and analyze of a software distribution. Fortunately it is as easy as running the Go build tool with a verbose flag, and all the intermediate build steps are shown as executable commands on the screen.

We have not performed any real measurements of the build speed gained from Go's dependency model. But it definitely feels very rapid. One example of this is that the whole Go distribution including C-compilation of gc, the Go tools and the Go compilation of the standard library takes about 15 wall clock seconds to compile (without the tests) on our machine[3].

When we wrote our first Go project we felt that the environment set up was forced upon us; why can we not decide how and where we want to place our Go project source files. Having workspaces makes us think about the clumsy usage of Eclipse workspaces which are cluttered with many hidden configuration files spread all over the project directories. Then we realized that we can have multiple workspaces in our $GOPATH so that we still can have coherent workspaces. After a while we started to think that having one standard way of structuring source files can only be good, to keep packages consistent and to work with all Go tools. There is a potential problem by having multiple Go workspaces though. If two packages are created with the same name but in different workspaces, the Go build tool use the first one found in $GOPATH. During some testing we were victim of this believed that our test program worked fine, but what we were really executing was the original unmodified module that was in another workspace. This is the same overlaying problem, but mostly seen as a feature, as with the POSIX PATH variables. We think that it would be convenient if the build tool at least would search and find all matching packages and warn if there are multiple packages with the same name.

3.2.2   gc & gccgo

The situation of the two compiler chains that can be used with the go build tool is both fortunate and troublesome. On the good side we get a broad range of supported architectures by having a GCC front-end for Go. Having two compilers is also good for the future development, much similar to why monopoly can be bad for development. As the Go developer Ian Lance Taylor says [123]:

> "Having two different implementations helps ensure that the spec is complete
> and correct: when the compilers disagree, we fix the spec, and change one or
> both compilers accordingly."

---

[3]Linux 3.2.0-4-amd64 #1 SMP Debian 3.2.57-3+deb7u1 x86_64 GNU/Linux, Intel i7-4770 3.4GHz, 16GiB memory.

Unfortunately it turns out that the development of gccgo lags behind the development of the gc compilers. For the past two releases of Go's language specification, gccgo has been implementing an older version. This is partly because the GCC front-end and GCC are different projects with uncoordinated release schedules. For example when Go 1.1 was released in May 2013 gccgo was released two months earlier in GCC version 4.8.0 and only implemented parts of the Go 1.1 specification [124] [125]. During the work with this thesis GCC 4.9.0 was released which has support for Go 1.2.1 while gc will very soon support 1.3 [126]. At the moment however, when the language is still young, the difference between two consecutive versions can be large and being behind on one update to the language specification can make it hard to use packages written for the newer version. This problem was experienced by us during our development of the door control system. We used a D-Bus wrapper library, *go.dbus*, which uses standard library functions[4] that was added at a later version than our gccgo supported [127]. This led us to the need of building a cross-compiling experimental version of gccgo.

As described before, the gc compilers can only produce statically linked executables. There are many opinions about static versus dynamic linking. Many people, including the Go author Rob Pike, are skeptical to the benefits of it [128]. Problems with dynamic linking include security issues (if a malicious library loaded) and mismatch between versions of shared objects on the system. Therefore we do not expect the gc compilers to support dynamic linking in the near future. When developing for embedded platforms however, it is common to use dynamic linking. To achieve this, the only option is currently to use gccgo as it supports both static and dynamic linking. Only having gccgo as an option is not a problem if there is no need to have the latest version of Go's language specification implemented, as gccgo lags behind.

However we did not experience everything to be good with compiling Go packages. It is already a common practice to treat compiler warnings as errors[5] both gc and gccgo takes this to the extreme and only have errors. It is a great idea that a compiler tool refuses to accept input source files that has unused imports and variables[6] when building for a release version; clearly code should not be shipped that is not used. It is maybe even good to be aware of this during the shorter development cycles so that it is known what code is actually used when debugging etc. Both gc and gccgo treats this as an error and refuses to accept such source files for compilations, with no built-in option for disabling this. This we often experienced as tedious when practicing small code-compile-cycles when frequently adding, removing and toggling small pieces of code. In those situations we wish there was a way of temporarily toggling off these errors checks, and make them warnings instead. On the other hand, that would possibly quickly lead to that option being included by default in an alias or build script and the compilation times would increase.

3.2.3   Cross-compilation

At the time writing, cross-compiling a Go program is a trivial task if the target is one of the architectures supported by the gc compilers. If cross-compilation to another architecture is something that is wanted, like in this thesis, the process in configuring and building a cross-

---

[4]bytes.TrimPrefix()

[5]Like the *-Werror* flag to GCC.

[6]Inconsistently does not report unused unexported global variable tho.

compiling build tool is badly documented and buggy. It was only until after we consulted the community that we were made aware of the procedure one should take in order to do this properly. All the problems that we have had with cross-compilation makes us believe that this is something that has not got as much attention as other things in the Go-project. To help Go in being practical to use for embedded programming, these issues should be solved and the process must be documented. We have tried ourselves to document the process, but it has not yet been verified by an internal Go member yet that this document is correct.

One good thing about the cross-compilation part is that building a cross-compiling GCC with support for Go is no harder than building a normal cross-GCC toolchain. Many companies and projects in this field have routines and scripts for this process already. All that is needed to support go is to add it to the list of supported languages in the final step of the GCC build. The bad part is that building a cross-compiling toolchain is a very hard problem to begin with. The part that makes cross-compilation hard with Go is the combination of using a cross-compiling gccgo with the Go tools. The lacking support of using compiler tools with triplet names and that the Go build has to be compiled in a special way to work for cross-compilation makes the whole process hard (until properly documented).

3.2.4   C-Integration

How well does integration between Go and C work in practice? We had some first-hand experience with this when we tried out the different features with toy examples and when we linked with a large C-library for communication over D-Bus, which is used in many Axis products, to our Go-code. While there was some bugs with the tools, it is clear that the developers have thought through the process and tried to make it as easy as compiling normal Go programs for example that C/C++ files found in the same directory as Go modules importing C are automatically compiled or that pkg-config can be used to resolve long import and linking flags.

The first problem we encountered with C-integration was about calling C-functions taking a variadic number of arguments, *varargs*, from Go as cgo does not support this. Go has support for variadic number of arguments, but it works differently from C so there is not automatic translation. A work-around is to create a C gateway function for each unique call to the C vararg function.

C in Go comments

One of the things we have been the most annoyed with concerning C-integration is that the directives for the cgo tool, and even C-code, are written in plain Go comments. While this keeps the Go syntax cleaner, we find that it makes C-integration worse. A reason, among others, is that all text editors we have come around when coding Go, treats Go comments as comments independent of context and content. This means that the C-code and compiler directives will not have syntax highlighting and is not supported by other text editor tools that works on code blocks. This means that writing C-code like this becomes very impractical. When building Go programs where inline C-code is found by the cgo tool, it is extracted to a file of its own and is compiled with a C compiler. If the compiler finds

and error or emits a warning it will be hard for the programmer to map that line number in the generated file to the erroneous line in the Go-file. Effectively the programmer would have to invoke the Go tool with an option to keep the temporary build directory and then manually inspect the generated C-files to find the errors more easily.

It is even error-prone to deal with Go-code using directives in comments. A very annoying and hard to find bug is when a Go function is exported but there is a mysterious linking problem that is hard to analyze if one does not know what is going on. In most programming languages, including Go, a start-of-comment symbol (// in Go) is by convention followed by a space and then the comments. However, the cgo export command requires that there should be no space between the slashes and the keyword i.e. *//export* is correct and *//␣export* is wrong. If the export functionality would have been included in the language, compilers and even text editors could have highlighted and warned about this common mistake.

A good thing about the annoyances of writing C-code in Go-comments is that it forces the programmer to write less C-code in Go source files, which is in line with our view: separate code from different programming languages. We would prefer to either put all C to Go interacting code in normal C files or that there would be a built-in language construct specifying that, what follows is C-code and not Go-code (similar to how C-code can be used in C++ source files with the *extern "C"* construct). A reason to have the C-code in a Go source file is that the code is (or should be) coherent to the Go-code. Coherent code should be close for easier understanding and maintenance. When we first tried out linking some C-libraries to Go programs we did not understand why we would want have C-code in Go source files. We wanted to have our C programs in C files only. It soon became clear to us that they are needed to deal with type conversions, macros, pointer conversion and callbacks. We think that, to keep Go programs clean, C-code in Go sources should be limited to the bridging between the languages and not implement any logic at all.

Memory management

There are some things to watch out for when integrating with C and one of them is memory related. While Go is garbage collected, C-code that Go might interact with is not. This means that the C-code have to keep track of its own memory deallocation as usual. Further, if data allocated by Go should be shared with C-code, it must be stored in a global variable. This extra thinking makes the integration a bit harder, but there is probably no way around it. While there are built-in functions for converting between Go strings and C null-terminated character arrays one must keep in mind that these functions makes copies of the data and not in place conversion. String-intensive programs should consider another representation that can be used in both C and Go without copying [82, Turning C arrays into Go slices]. During our own project, we forgot to null-terminate a Go string that was passed to a C-library which caused mysterious and random errors at other places in the code. This means that seamless interaction with C-code is not possible. One must always keep in mind the differences between the languages and think about memory and types.

Building

A build of a Go package that integrates with C takes a notable longer time to build. This is not surprising as much more automatic work is done: C-code is extracted to new files, Go exports generates C sources and headers, more C/C++ files are searched for in the package directory, C compilers are invoked, C programs are linked to libraries, etc. The positive side of this is that we can avoid having to write own build instructions (apart from the C compiler directives) and the C-integration becomes seamless with the build of plain Go packages. The downside is the builds being slowed down. We think that in the long run, the automation will save a lot of time for developers since they do not have to spend time on figuring out the right steps in compiling and linking together Go and C-code. This outweighs the loss in build time.

Improvements

We think that overall the C-integration works well for small and easy code but becomes complicated in more complex integrations with libraries that use callbacks and many newly defined types. Ideally, a programmer that knows Go and C would be able to write Go programs interacting with C and the other way around without knowing more. Currently though, the programmer must also alway have in mind how to integrate between the languages. We believe that there are room for improvements here. Hopefully it is possible to simply the integration so that less wrapper code has to be written.

3.3 Development Tools

3.3.1 Package manager

For Go to be practical to use for rapid development, it must be easy to manage packages. Go's package manager works great but we think it could be even better. Currently the head of a source code version control system is checked out when the Go tool fetches the packages. This can make it confusing which version is in use. The Go developer Andrew Gerrand thinks this is a release engineering problem, that the Go packages should have a consistent API that is not broken between versions. If a break in the API is introduced, then the import path of the package should change as well i.e. adding a version number to the path [129, 37:35]. An alternative would be to specify the version, for example a git tag or changeset, which should be used.

3.3.2 Documentation & Testing

As discussed earlier the goal with Go is to minimize the efforts needed by programmers. This is something that is reflected in the built-in tool for documentation and testing as well. Having these tools built into the language is nice because everything works "out of the box" which means that developers do not have to download or install anything. They are easy to use and come with great functionalities, more thoroughly described in Section 2.3. Furthermore, having composition over inheritance in Go is something that

help both of these tools. Composition encourages the creation of smaller interfaces with implementations that are easy to test and document.

Writing documentation and not having to follow a specific syntax, unlike Java for example, can in many cases results in more descriptive and understandable text. This means that companies themselves can set up policies for how documentation should be done, which is positive in one way. It could also be bad since it could result in a lot of different standards being used. A convention has already been formed; the comment is started with the name of the element that is documented. We think this is not the right approach. The reason for this is that if or when the name of an element is changed, the same change must be done for the documentation which is something that easily can be forgotten or missed. As it is right now, this is an important part of the generation of documentation [99].

3.3.3 Debugging

The state of debugging Go programs is a very tricky situation. The way debugging is done right now works for many cases. However, as explained before, it is not an option for embedded programming when we need to do remote debugging or inspect core dumps. In this case a debugger is an important tool for locating the errors, if it cannot be directly found in the erroneous code that is. Unless GDB can be modified to understand goroutines it will probably take a while until we see a fully functional Go debugger especially if we listen to what Rob Pike said at GopherCon in April 2014: "There are plans for a debugger but they are very sketchy. It is hard because the operating systems today do not want you to write a debugger." [130, 46:18]. This makes us believe that it will take a while until the debugging situation becomes better for Go programmers.

3.4 Community

During the whole thesis work we have been in contact with the Go community. When learning about the language and its tools the greatest source for information, apart from the official documentation, is the user mailing list *golang-nuts*. This is a open mailing list with participants ranging from complete beginners, experts to famous Go developers. We have seen that many of the most influential Go developers are very active in the user community as they have participated in many of the threads we started ourselves. Having the Go developers at such close range makes the gap between the users of Go and its developers very short. This gives the Go developers better feedback on the language and development, and users of the language get help with their problems or can express their opinions about the future of the language. The mailing list is not the only place where Go discussions take place. We have been involved in the website Stack Overflow, which is a community for asking and answering questions related to programming. There, we have followed the discussions about Go and answered several questions ourselves [131].

Having a close and helpful community makes it easier to propose patches for bugs or propose enhancements for the tools. The fact that many of the Go tools, and soon the gc compilers, themselves are written in Go makes it easier to contribute as one does not have to know another language. Having a self-contained language also makes a good quality assurance reference of the language. Anyone can contribute to the Go-project,

which includes both the language and the surrounding tools. If the contribution is in form of a bug-fix or patch it can be posted on the Go project website[7]. If the contribution includes a design change of something, it is suggested that a thread on the official mailing list is opened in order for the design to be verified before anything is implemented. When the change request is submitted, it has to be accepted by one of the project administrators. A more detailed description on how contributions can be made to the project can be found on Go's website [49].

If the Go user base becomes much larger, it is unclear how the good things about the community will be affected. If the mailing lists becomes flooded with too much activity, maybe the important Go developers will lose interest in being active in the mailing list or be unable to pay attention to every thread that is created.

3.5   The Future of Go

The fact that the development of Go takes place in the open makes it possible for everyone to participate in the discussion and see what direction the language is taking. Looking at the upcoming 1.3 release of Go it looks like the pace of introducing features is slow, as no languages changes are coming and none are proposed for future versions (from what we have found). This is in line with the Go philosophy of having a simple and well afterthought language. The 1.3 release has, apart from numerous bug fixes, focus on performance. A couple of the changes are: the change of stack model that makes them initially a little bit more expensive but more effective in the long run, speed up of the linking step, garbage optimizations and more [132]. This is a sign that the project is stabilizing before considering adding new features. We have not found a long-term road map but one thing we have seen talk about for versions beyond 1.3 is the translation and rewriting of the gc compilers from C to Go [63] [133]. Maybe we will see the return of network channels or the introduction of generics in the future of Go?

We have previously discussed if Go is suited for embedded development but there are more factors that matter in Go's success in this area; the community. The question is if there will be a large enough amount of developers in the community to push the development forward of the cross-compilation tools. If not, will the Go-team drop the development of these parts and the focus on being a general-purpose language? Discontinuation is something that we are familiar with when it comes to Google products that do not have enough users or make enough money [134] [135]. This is however not only a concern that targets embedded programming with Go but also the whole Go-project. What happens to Go if Google is not interested in it anymore? We think that this will not happen in the near future because Google uses Go themselves in many of their internal services. But if it does, our prediction is that the language will live on through its community and its very dedicated creators since it is a open project with open sourced tools. How well the quality of the language would be maintained without Google is hard to say. This aspect strongly depends on the actions the community take when after the (unlikely) discontinuation of Go.

There are more factors that play a role in Go's success in the future, other languages is one example. Every now and then, new languages are created and the question is if Go

---

[7]code.google.com/p/go

will be able to keep its users from switching from Go. An example of a language that often comes up when discussing adoption of Go is the *Rust* programming language developed by Mozilla. It is a compiled system programming language that has focus on running fast, having garbage collection (optional), concurrency, type inference, C-bindings, etc. and is currently in version 0.10 (alpha). Although these features sound very familiar, Rust also includes the feature of generics. This could be a reason to why developers would choose Rust over Go, especially the ones that do not like the approach that the Go-team have taken to solve the same problems that generics solves (through interfaces and type assertion) [136] [137]. Currently Rust is very young and undergoes rapid development and many changes not making it a viable alternative for production at this moment. In a near future we think that both Go and Rust will be two of the most popular modern languages that are adopted for both personal and corporate usage.

Throughout this thesis, we have talked about cross-compiling Go-code and the difficulties with it. There are other compilers out there for example Clang/LLVM that could possibly easier to get to work for cross-compilation, since "Clang/LLVM is natively a cross-compiler" [138] [139]. However, the compiler is not enough and the problem of getting all the needed header files and utilities for the target still exists. Currently there is an experimental project building a Go front-end for LLVM [140]. If this front-end evolves to a stable project, cross-compilation for Go programs would probably become easier.

3.6 Reviewing a Programming Language

In retrospect, was it meaningful to spend such a long time of the thesis work to implement the physical access control system? That time could have been spent on doing language performances or study of the language more directly. We think that it was a really good idea to have a project that we wanted to complete. The reason behind this is that the project forced us to try out different things that we might have ignored or overlooked otherwise, for example the steps of cross-compiling. We could have just made a minimal working example and come to the conclusion that it works. The fact that we had a larger project exposed us to many details that actually matters for practical development. When we wanted to use a D-Bus language binding library, it required a more recent version of Go. Since no cross-compiler builder tool was found that could produce one we were forced to learn about that process ourselves, to be able to continue with our project[8].

During the project we took the opportunity to try to use as many Go features as possible and designed a module that internally used goroutines and channels extensively to get the work done in a simple and understandable way. Through the same module we exposed ourselves with different kinds of testing and documentation. One of the risks in taking on a large project is that it consumes too much time. There were several occasions where we had to take a step back and consider what work we actually would learn something from.

---

[8]Of course we could have taken the easy way of just modifying the library, but we were set out to learn things, not to take the easy way.

Chapter 4

Conclusions

4.1 Summary

This master's thesis was set out to find out how good Go is and specifically if it is suitable and ready to be used for embedded programming. We think that Go's approach for simplicity and encouragement for good programming practices makes it very useful for large-scale software development. Many languages have shorthands, complex constructs and doubtful concepts that are intended to make it easier for the programmer. However, we think that the lack of such features in Go makes software development better in the long run. We think that building concurrency primitives into the language fits very well with how we want to organize software. The fact that Go is easy to learn makes it very viable to be adopted. However, more complex integration with C-code is an area in Go that includes many pitfalls and is therefore something that requires more study before it can be utilized efficiently. Even though this area works well in simpler cases, we believe that it could be done more smoothly.

While the language is good and the tools are splendid for the most part, the state of cross-compilation is worse. Cross-compilation is a tricky problem to start with but the Go build tool could have worked better with gccgo than it does today. We found that there are people doing cross-compilation with Go, but they are few compared to the rest, meaning that this subject receives little attention in the community. As the Go user base grows the interest for these tools will grow, but we think it will take a while since this subject is not prioritized. If Go is to be used for embedded programming today, our recommendation is to wait for the tools to become mature or be prepared to spend time on working around the quirks and fixing the build tools.

We can recommend people who want to review a programming language to do so by implementing a larger project in that language. If the project is relevant for what is to be investigated, it will help to reveal language features and concepts, give a more realistic view of the language and show how tools really work in practice.

4.2 Future Research

During the scope of this thesis we had to limit ourselves to able to complete the task within a reasonable time frame. We stumbled upon many subjects that we would love to study further but were impossible to fit into this thesis. We would for example like to study the effectiveness of goroutines compared to OS threads, both in theory and in practice. Things to study include the overhead of the Go runtime and extra CPU and memory footprints compared to the cost of setting up OS threads. It would also be interesting to do a comparison of the optimization and compilation speeds between gc and gccgo. Although we have looked at Debian's language comparison we would have liked to do the comparison ourselves on the MIPS device that we had access to. The subject of debugging is still unresolved in Go. Future research could include finding out how to get GDB to work properly with Go or how to write a new debugger that supports Go specifics like goroutines. For embedded programming, it would be interesting to study how memory profiling with tools like Valgrind can be done, especially when integrating with C-code.

If we would have had more time, would have liked to do a rigorous comparison of the original PACS software and our re-implementation in terms of execution performance, understandability, ease of development and more.

Originally our goal with this thesis was two-fold; to investigate Go but also that this thesis could serve as a reference in how to review a programming language. However, under this thesis work we realized that producing a reference template on this subject would be very time consuming and could serve as a master's thesis on its own. Doing such work is still interesting and is something that we would like to as future research if time allows.

Chapter 5
Bibliography

[1] "Documentation - the go programming language," accessed 2014-04-11. [Online]. Available: http://golang.org/doc/

[2] L. Flon, "On research in structured programming," *SIGPLAN Not.*, vol. 10, no. 10, pp. 16–17, Oct. 1975. [Online]. Available: http://doi.acm.org/10.1145/987253. 987256

[3] "Quotes about go programming," accessed 2014-05-15. [Online]. Available: http://go-lang.cat-v.org/quotes

[4] "Go at google: Language design in the service of software engineering," accessed 2014-01-16. [Online]. Available: http://talks.golang.org/2012/splash.article

[5] "Concurrency is not parallelism," accessed 2014-04-16. [Online]. Available: http://concur.rspace.googlecode.com/hg/talk/concur.html

[6] "Expressiveness of go," accessed 2014-04-24. [Online]. Available: http://talks. golang.org/2010/ExpressivenessOfGo-2010.pdf

[7] I. L. Taylor, "The go frontend for gcc," 2010, accessed 2014-04-24. [Online]. Available: http://talks.golang.org/2010/gofrontend-gcc-summit-2010.pdf

[8] "Gccgo in gcc 4.7.1," accessed 2014-05-02. [Online]. Available: http://blog. golang.org/gccgo-in-gcc-471

[9] "Go faq," accessed 2014-04-28. [Online]. Available: http://golang.org/doc/faq

[10] "Dynamic linking: Advantages and disadvantages," accessed 2014-04-16. [Online]. Available: https://www.informit.com/guides/content.aspx?g=cplusplus& seqNum=152

[11] "platform on foldoc," accessed 2014-04-23. [Online]. Available: http://foldoc.org/platform

[12] "automake: Cross-compilation," accessed 2014-04-23. [Online]. Available: http://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Cross_002dCompilation.html

[13] "What is ctags?" accessed 2014-01-29. [Online]. Available: http://ctags.sourceforge.net/whatis.html

[14] "Cscope home page," accessed 2014-01-29. [Online]. Available: http://cscope.sourceforge.net/

[15] ISO, "Software engineering – product quality – part 1: Quality model," *ISO*, 2001, accessed 2014-05-02. [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749

[16] ——, "Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models," *ISO*, p. 34, 2011, accessed 2014-05-02. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=35733

[17] H.-C. N. Olga Antropova, Lisa Hollermann and P. Sharma, "Comparing programming languages, www presentation," 1997. [Online]. Available: http://www.eecs.ucf.edu/~leavens/ComS541Fall97/hw-pages/comparing/

[18] G. H. Pradeep Bollineni, Patrick Hartling, "Comparing programming languages," 1998. [Online]. Available: http://www.eecs.ucf.edu/~leavens/ComS541Fall98/hw-pages/comparing/

[19] "C++ keywords," accessed 2014-05-02. [Online]. Available: http://en.cppreference.com/w/cpp/keyword

[20] "The go programming language specification," accessed 2014-04-22. [Online]. Available: http://golang.org/ref/spec

[21] "A tour of go," accessed 2014-05-02. [Online]. Available: http://tour.golang.org/

[22] B. Kernighan and D. Ritchie, *The C Programming Language*. Prentice Hall, 1988.

[23] J. Lewis, *Java software solutions : foundations of program design*. Boston: Pearson/Addison Wesley, 2005.

[24] "Gobject reference manual," accessed 2014-04-25. [Online]. Available: https://developer.gnome.org/gobject/stable/pr01.html

[25] "Bjarne stroustrup's c++ glossary," accessed 2014-04-25. [Online]. Available: http://www.stroustrup.com/glossary.html#Gpolymorphism

[26] "Golang fmt," accessed 2014-04-25. [Online]. Available: http://golang.org/pkg/fmt/

[27] "Duck typing: Ruby study notes," accessed 2014-04-25. [Online]. Available: http://rubylearning.com/satishtalim/duck_typing.html

[28] "Effective go," accessed 2014-04-17. [Online]. Available: http://golang.org/doc/effective_go.html

[29] K. Knoernschild, *Java design : objects, UML, and process*. Boston, MA: Addison-Wesley, 2002.

[30] "Google i/o 2010 - go programming," accessed 2014-04-25. [Online]. Available: https://www.youtube.com/watch?v=jgVhBThJdXc&t=11m12s

[31] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle, "Mixins in strongtalk," 2002, accessed 2014-04-25. [Online]. Available: http://www.bracha.org/mixins-paper.pdf

[32] A. Anastasopoulos, "Contiguous stacks in go," accessed 2014-04-17. [Online]. Available: http://agis.io/2014/03/25/contiguous-stacks-in-go.html

[33] "Contiguous stacks," accessed 2014-04-17. [Online]. Available: https://docs.google.com/document/d/1wAaf1rYoM4S4gtnPh0zOlGzWtrZFQ5suE8qr2sD8uWQ/pub

[34] B. Anderson, "[rust-dev] abandoning segmented stacks in rust." [Online]. Available: https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html

[35] "Go language patterns – semaphores," accessed 2014-04-22. [Online]. Available: http://www.golangpatterns.info/concurrency/semaphores

[36] "Gophercon 2014 opening keynote by rob pike," accessed 2014-05-12. [Online]. Available: http://youtu.be/VoS7DsT1rdM?t=48m

[37] "[golang-nuts] new netchan," accessed 2014-05-12. [Online]. Available: https://groups.google.com/forum/#!searchin/golang-nuts/netchan/golang-nuts/Er3TetntSmg/YExbTtIH9jYJ

[38] "Java collections framework overview," accessed 2014-04-28. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html

[39] "<algorithm>; - c++ reference," accessed 2014-04-28. [Online]. Available: http://www.cplusplus.com/reference/algorithm/

[40] "crypto/tls," accessed 2014-05-12. [Online]. Available: http://golang.org/pkg/crypto/tls/

[41] "[golang-nuts] go is production ready, but the crypto/tls package isn't?" accessed 2014-05-12. [Online]. Available: https://groups.google.com/forum/#!topic/golang-nuts/LjhVww0TQi4

[42] "The java tutorials > generics," accessed 2014-04-28. [Online]. Available: http://docs.oracle.com/javase/tutorial/extra/generics/intro.html

[43] "gen - a generics library for go," accessed 2014-04-28. [Online]. Available: http://clipperhouse.github.io/gen/

[44] "Go license," accessed 2014-05-11. [Online]. Available: http://golang.org/LICENSE

[45] "Go patents," accessed 2014-05-11. [Online]. Available: http://golang.org/PATENTS

[46] "gccgo license," accessed 2014-05-12. [Online]. Available: http://code.google.com/p/gofrontend/source/browse/LICENSE

[47] "golang-dev," accessed 2014-05-11. [Online]. Available: https://groups.google.com/forum/#!forum/golang-dev

[48] "Go issues," accessed 2014-05-11. [Online]. Available: https://code.google.com/p/go/issues/list

[49] "Go contribution guidelines," accessed 2014-05-11. [Online]. Available: http://golang.org/doc/contribute.html

[50] "Google individual contributor license agreement, v1.1," accessed 2014-05-11. [Online]. Available: https://developers.google.com/open-source/cla/individual?csw=1

[51] "Installing go from source," accessed 2014-04-25. [Online]. Available: http://golang.org/doc/install/source

[52] "Go in debian," accessed 2014-04-15. [Online]. Available: https://packages.debian.org/wheezy/golang

[53] "Arch linux go," accessed 2014-04-15. [Online]. Available: https://www.archlinux.org/packages/community/i686/go/

[54] "Setting up and using gccgo," accessed 2014-01-13. [Online]. Available: http://golang.org/doc/install/gccgo

[55] "Ros tutorials navigatingthefilesystem," accessed 2014-04-15. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem

[56] "Environment variables," accessed 2014-05-02. [Online]. Available: http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap08.html#tag_08_03

[57] "How to write go code - the go programming language," accessed 2014-04-15. [Online]. Available: http://golang.org/doc/code.html

[58] "cat-v go utils and tools," accessed 2014-04-15. [Online]. Available: http://go-lang.cat-v.org/dev-utils

[59] "Github laher/goxc," accessed 2014-04-15. [Online]. Available: https://github.com/laher/goxc

[60] "gc - the go programming language," accessed 2014-01-28. [Online]. Available: http://golang.org/cmd/gc/

[61] R. Pike, "How to use the plan 9 c compiler." [Online]. Available: http://plan9.bell-labs.com/sys/doc/comp.pdf

[62] K. Thompson, "Plan 9 c compilers." [Online]. Available: http://plan9.bell-labs.com/sys/doc/compiler.pdf

[63] A. Gerrand, "Toward go 1.3," accessed 2014-04-17. [Online]. Available: http://talks.golang.org/2014/go1.3.slide#19

[64] "[golang-nuts] shared libraries and dynamic loading?" accessed 2014-04-10. [Online]. Available: https://groups.google.com/forum/#!topic/golang-nuts/8xP5dKf0LJc

[65] "go - gccgo -static vs -static-libgo - stack overflow," posted 2014-02-05. [Online]. Available: http://stackoverflow.com/questions/20369106/gccgo-static-vs-static-libgo/22011658

[66] "Gcc license," accessed 2014-05-15. [Online]. Available: http://repo.or.cz/w/official-gcc.git/blob_plain/HEAD:/COPYING

[67] "Llvm license," accessed 2014-05-15. [Online]. Available: http://clang.llvm.org/features.html#license

[68] "An introduction to cross compilation with go 1.1," accessed 2014-04-22. [Online]. Available: http://dave.cheney.net/2013/07/09/an-introduction-to-cross-compilation-with-go-1-1

[69] "Installing gcc: Configuration - gnu project - free software foundation (fsf)," accessed 2014-04-25. [Online]. Available: http://gcc.gnu.org/install/configure.html#TOC1

[70] "crosstool-ng," accessed 2014-02-17. [Online]. Available: http://crosstool-ng.org/

[71] "Gentoo creating a cross-compiler," accessed 2014-04-23. [Online]. Available: https://www.gentoo.org/proj/en/base/embedded/handbook/?part=1&chap=2

[72] E. Westrup, "ewxb - erik westrup's gcc cross-compiler builder," posted 2014-03-27. [Online]. Available: https://github.com/erikw/ewxb_gcc_cross-compiler_builder

[73] ——, "[golang-nuts] simplification of mips cross-compilation?" posted 2014-02-04. [Online]. Available: https://groups.google.com/forum/#!topic/golang-nuts/PgyS2yoO2jM

[74] "Specifying target triplets - autoconf," accessed 2014-04-09. [Online]. Available: http://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.69/html_node/Specifying-Target-Triplets.html

[75] E. Westrup, "Gccgocrosscompilation symlinks," posted 2014-02-20. [Online]. Available: http://code.google.com/p/go-wiki/wiki/GccgoCrossCompilation# Symlink

[76] ——, "code review 63200044: go, cgo: Added 'go build' and cgo option "-gccgonoarcha"," posted 2014-02-13. [Online]. Available: https://codereview. appspot.com/63200044/

[77] ——, "code review 65390043: go, cgo: Added support for goarch=mips," accessed 2014-02-18. [Online]. Available: https://codereview.appspot.com/65390043

[78] ——, "Issue 7398: cmd/cgo: unknown ptrsize for $goarch mips when using gccgo," posted 2014-02-24. [Online]. Available: https://code.google.com/p/go/ issues/detail?id=7398

[79] ——, "Gccgocrosscompilation," posted 2014-02-20. [Online]. Available: http://code.google.com/p/go-wiki/wiki/GccgoCrossCompilation?ts= 1392894053&updated=GccgoCrossCompilation

[80] "cgo - the go programming language," accessed 2014-01-28. [Online]. Available: http://golang.org/cmd/cgo/

[81] "pkg-config," accessed 2014-04-16. [Online]. Available: http://www.freedesktop. org/wiki/Software/pkg-config/

[82] "Cgo - go-wiki," accessed 2014-04-16. [Online]. Available: http://code.google. com/p/go-wiki/wiki/cgo

[83] F. Pettersson, "Issue 7573: cmd/cgo: undefined reference when linking a c-library with gccgo," posted 2014-03-18. [Online]. Available: http://code.google.com/p/ go/issues/detail?id=7573

[84] E. Westrup, "code review 80780043: cmd/go: Use exported cgoldflags when compiler=gccgo," posted 2014-03-26. [Online]. Available: https://codereview. appspot.com/80780043/

[85] ——, "cmd/go: Use exported cgoldflags when compiler=gccgo," posted 2014-03-26. [Online]. Available: http://code.google.com/p/go/source/detail?r= 0134c7020c40a2728bac62899cd7c36a4381e5c8

[86] "How can i wrap zlib in golang?" accessed 2014-05-08. [Online]. Available: http://stackoverflow.com/questions/14686256/how-can-i-wrap-zlib-in-golang

[87] "runtime: allow cgo callbacks on non-go threads," accessed 2014-05-29. [Online]. Available: http://code.google.com/p/go/source/detail?r=1d5a80b07916

[88] "The cover story," accessed 2014-02-19. [Online]. Available: http://blog.golang. org/cover

[89] "stretchr/testify · github," accessed 2014-02-19. [Online]. Available: https: //github.com/stretchr/testify/

[90] "Ginkgo," accessed 2014-02-19. [Online]. Available: http://onsi.github.io/ginkgo/

[91] "Gomega," accessed 2014-02-24. [Online]. Available: http://onsi.github.io/gomega/

[92] "Debugging go code with gdb," accessed 2014-05-06. [Online]. Available: http://golang.org/doc/gdb

[93] M. J. Eager, "Introduction to the dwarf debugging format," 2007, accessed 2014-05-06. [Online]. Available: http://www.dwarfstd.org/doc/Debugging%20using%20DWARF.pdf

[94] "Gdb: The gnu project debugger," accessed 2014-05-06. [Online]. Available: https://www.sourceware.org/gdb/

[95] E. Westrup, "Issue 7506: cmd/6l: No .debug_gdb_script section in elf," posted 2014-03-11. [Online]. Available: https://code.google.com/p/go/issues/detail?id=7506

[96] ——, "Issue 7796: misc/gdb: info goroutines -> python exception "attempt to extract a component of a value that is not a (null)"," posted 2014-04-16. [Online]. Available: http://code.google.com/p/go/issues/detail?id=7796

[97] ——, "Gdb: View goroutine state in core dump," posted 2014-02-11. [Online]. Available: https://groups.google.com/forum/#!topic/golang-nuts/YkssIznjfHU

[98] "[golang-dev] fwd: The state of gdb support," posted 2014-03-05. [Online]. Available: https://groups.google.com/forum/?hl=de#!searchin/golang-dev/gdb/golang-dev/UiVP6F-9-yg/lqS3sbyfTZMJ

[99] "Godoc: documenting go code - the go blog," accessed 2014-02-25. [Online]. Available: http://http://blog.golang.org/godoc-documenting-go-code/

[100] "indent(1)," accessed 2014-04-29. [Online]. Available: http://linux.die.net/man/1/indent

[101] "gofmt," accessed 2014-04-29. [Online]. Available: http://golang.org/cmd/gofmt/

[102] "An autocompletion daemon for the go programming language," accessed 2014-04-22. [Online]. Available: https://github.com/nsf/gocode

[103] "parser - the go programming language," accessed 2014-04-14. [Online]. Available: http://golang.org/pkg/go/parser/

[104] "lint(1)," accessed 2014-04-14. [Online]. Available: http://www.unix.com/man-page/FreeBSD/1/lint

[105] "golang/lint," accessed 2014-04-14. [Online]. Available: https://github.com/golang/lint

[106] "Command go," accessed 2014-04-24. [Online]. Available: http://golang.org/cmd/go/

[107] "Physical access control | axis communications," accessed 2014-05-05. [Online]. Available: http://www.axis.com/corporate/press/us/releases/viewstory.php?case_id=3130

[108] "Axis communications enters the physical access control market," accessed 2014-02-05. [Online]. Available: http://www.axis.com/access_control/

[109] A. Communications, "Ip opens doors to a new world of physical access control," Axis, Tech. Rep., 2013. [Online]. Available: http://www.axis.com/files/whitepaper/wp_pacs_ip_benefits_53812_en_1309_hi.pdf

[110] "D-bus," accessed 2014-05-05. [Online]. Available: http://www.freedesktop.org/wiki/Software/dbus/

[111] "Glib," accessed 2014-05-05. [Online]. Available: https://developer.gnome.org/glib/

[112] "Less is exponentially more," accessed 2014-05-08. [Online]. Available: http://commandcenter.blogspot.se/2012/06/less-is-exponentially-more.html

[113] "The computer language benchmarks game," accessed 2014-04-28. [Online]. Available: http://benchmarksgame.alioth.debian.org/

[114] "Floss weekly 284," accessed 2014-05-07. [Online]. Available: http://twit.tv/show/floss-weekly/284

[115] "Issue 5328: syscall: constant epollet inconsistent with /usr/include/sys/epoll.h on linux," accessed 2014-05-08. [Online]. Available: http://code.google.com/p/go/issues/detail?id=5328

[116] E. Gamma, *Design patterns : elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995.

[117] "container/list," accessed 2014-05-09. [Online]. Available: http://golang.org/pkg/container/list/

[118] "pthread_cond_wait(3)," accessed 2014-04-14. [Online]. Available: http://linux.die.net/man/3/pthread_cond_wait

[119] "Ømq - the guide - Ømq - the guide," accessed 2014-04-14. [Online]. Available: http://zguide.zeromq.org/page:all

[120] "Java synchronized methods," accessed 2014-04-14. [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

[121] "Introducing the go race detector - the go blog," accessed 2014-04-14. [Online]. Available: http://blog.golang.org/race-detector

[122] "Make generating prerequisites automatically," accessed 2014-05-13. [Online]. Available: https://www.gnu.org/software/make/manual/html_node/Automatic-Prerequisites.html

[123] "Gccgo in gcc 4.7.1," accessed 2014-02-25. [Online]. Available: http://blog.golang.org/gccgo-in-gcc-471

[124] "Go 1.1 is released," accessed 2014-02-25. [Online]. Available: http://blog.golang.org/go-11-is-released.article

[125] "Go 1.1 release notes," accessed 2014-02-25. [Online]. Available: http://golang.org/doc/go1.1#gccgo

[126] "Gcc 4.9 release series," accessed 2014-04-23. [Online]. Available: http://gcc.gnu.org/gcc-4.9/changes.html

[127] "package dbus," accessed 2014-02-25. [Online]. Available: https://godoc.org/github.com/guelfey/go.dbus

[128] "Cat-v dynamic linking," accessed 2014-05-13. [Online]. Available: http://harmful.cat-v.org/software/dynamic-linking/

[129] "Google i/o 2012 - meet the go team," accessed 2014-05-13. [Online]. Available: https://www.youtube.com/watch?v=sln-gJaURzk

[130] "Opening day keynote," accessed 2014-05-09. [Online]. Available: http://confreaks.com/videos/3419-gophercon2014-opening-day-keynote

[131] "User erik westrup - stack overflow," accessed 2014-05-11. [Online]. Available: https://stackoverflow.com/users/265508/erik-westrup?tab=answers

[132] "Go 1.3 release notes," accessed 2014-05-12. [Online]. Available: http://tip.golang.org/doc/go1.3

[133] R. Cox, "Go 1.3+ compiler overhaul," accessed 2014-05-12. [Online]. Available: https://docs.google.com/document/d/1P3BLR31VA8cvLJLfMibSuTdwTuF7WWLux71CYD0eeD8/edit

[134] "A second spring of cleaning," accessed 2014-05-13. [Online]. Available: http://googleblog.blogspot.com.au/2013/03/a-second-spring-of-cleaning.html

[135] "Spring cleaning in summer," accessed 2014-05-13. [Online]. Available: http://googleblog.blogspot.se/2012/07/spring-cleaning-in-summer.html

[136] "Rust," accessed 2014-05-13. [Online]. Available: http://www.rust-lang.org/

[137] "The rust language tutorial," accessed 2014-05-13. [Online]. Available: http://static.rust-lang.org/doc/master/tutorial.html#generics

[138] "Cross-compilation using clang - clang 3.5 documentation," accessed 2014-04-10. [Online]. Available: http://clang.llvm.org/docs/CrossCompilation.html

[139] "Gcc github bergus/metta/build_toolchain.sh," accessed 2014-04-10. [Online]. Available: https://github.com/berkus/metta/blob/develop/build_toolchain.sh

[140] "Github - axw/llgo," accessed 2014-04-10. [Online]. Available: https://github.com/axw/llgo

Appendices

Appendix A

Code

A.1 gomips

Listing A.1: gomips

```bash
#!/usr/bin/env bash
# Wrapper for go using our gcc cross-compiler.

scriptname=${0##*/}
read -r -d '' usage <<EOF
Go MIPS builder!

Usage: ${scriptname} [-d] [-u | -U upload_destination] [ -f
    gccgoflags ] gofile | [-h | -?]"
  -d      Link dynamically (static is default).
  -u      Upload binary to default gyup destination.
  -U      Upload to specified destination with gyup.
  -f      Extra gccgoflags to pass to -gccgoflags.
  -h, -?     This help text.
EOF
upl_dest=""
gccgoflags=""
link_static="true"

if [ "$#" -eq 0 ]; then
  echo "$usage"
  exit 2
fi
while getopts "duU:f:h?" opt; do
  case "$opt" in
      d) link_static="false";;
      u) upl_dest="default";;
      U) upl_dest="$OPTARG";;
```

```
28      f) gccgoflags+=" $OPTARG";;
29    :) echo "Option -$OPTARG requires an argument." >&2;
         exit 1;;
30    h|?|*) echo "$usage"; exit 0;;
31  esac
32 done
33 shift $(($OPTIND - 1))
34
35 if [ "$link_static" == "true" ]; then
36   gccgoflags+=" -static"
37 fi
38
39
40 if [ "$scriptname" == "mgomips" ]; then
41   source "$(dirname $(realpath "$0"))/x_environment.sh"
42   export PATH="$TOOLS/bin:$PATH" # Has overlay gccgo binary
         that cross compiles.
43   export GOOS="linux"
44   export GOARCH="mips"
45
46   mgo build -compiler gccgo -gccgoflags "$gccgoflags" "$@"
47   ecode="$?"
48   if [ "$ecode" -ne 0 ]; then
49     printf "go build failed: %s\n" "$gocmd" >&2
50     exit "$ecode"
51   fi
52 elif [ "$scriptname" == "gomips" ]; then
53   WORK="/tmp/go-build$$"
54   mkdir "$WORK"
55
56   #export PATH="$HOME/bin/mipsel-unknown-linux-gnu/bin:
         $PATH" # Has overlay gccgo binary that cross compiles.
57   source "$(dirname $(realpath "$0"))/x_environment.sh"
58   export PATH="$TOOLS/bin:$PATH" # Has overlay gccgo binary
         that cross compiles.
59   export DISABLEARCH="yesyesyes"  # Custom built cgo that
         does not produce "-m32" or "-m64" options to gcc
         commandline.
60
61   mkdir -p $WORK/command-line-arguments/_obj/
62
63   gocmd=$(go build -compiler gccgo -gccgoflags "$gccgoflags
         " -n "$@" 2>&1)
64   ecode="$?"
65   if [ "$ecode" -ne 0 ]; then
66     printf "go build failed: %s\n" "$gocmd" >&2
67     exit "$ecode"
68   fi
69
70   cgo_flags=$(echo $gocmd | grep -Pzoi 'CGO_LDFLAGS=".*?
         (?=[^"])' | sed -e "s/\"\s\s*\"/ /g")
71   eval "export $cgo_flags" &>/dev/null
72   gocmd=$(echo "$gocmd" | sed -e 's/-m64//g' -e 's/-(/-\\(/
         g' -e 's/-)/-\\)/g' -e 's/\"\s\s*\"/ /g')
73
74   set -e
```

```
75    eval "$gocmd"
76
77    rm -r $WORK
78  else
79    echo "Unsupported version of script: $scriptname"
80  fi
81
82
83  if [ -n "$upl_dest" ]; then
84    binary="$(echo "$@" | sed 's/\.go//')"
85    if [ "$upl_dest" == "default" ]; then
86      gyup "$binary"
87    else
88      gyup -t "$upl_dest" "$binary"
89    fi
90  fi
```

A.2   C Function Pointer Callbacks

Listing A.2: Compiling and running

```
1  $ gcc -c clibrary.c
2  $ ar cru libclibrary.a clibrary.o
3  $ go build ccallbacks
4  $ ./ccallbacks
5  Go.main(): calling C function with callback to us
6  C.some_c_func(): calling callback with arg = 2
7  C.callOnMeGo_cgo(): called with arg = 2
8  Go.callOnMeGo(): called with arg = 2
9  C.some_c_func(): callback responded with 3
```

Listing A.3: goprog.go

```
1  package main
2
3  /*
4  #cgo CFLAGS: -I .
5  #cgo LDFLAGS: -L . -lclibrary
6
7  #include "clibrary.h"
8
9  int callOnMeGo_cgo(int in); // Forward declaration.
10  */
11  import "C"
12
13  import (
14    "fmt"
15    "unsafe"
16  )
17
18  //export callOnMeGo
```

```go
19  func callOnMeGo(in int) int {
20      fmt.Printf("Go.callOnMeGo(): called with arg = %d\n", in)
21      return in + 1
22  }
23
24  func main() {
25      fmt.Printf("Go.main(): calling C function with callback
            to us\n")
26      C.some_c_func((C.callback_fcn)(unsafe.Pointer(C.
            callOnMeGo_cgo)))
27  }
```

Listing A.4: cfuncs.go

```go
1   package main
2
3   /*
4
5   #include <stdio.h>
6
7   // The gateway function
8   int callOnMeGo_cgo(int in)
9   {
10          printf("C.callOnMeGo_cgo(): called with arg = %d\n
                ", in);
11          return callOnMeGo(in);
12  }
13  */
14  import "C"
```

Listing A.5: clibrary.h

```c
1   #ifndef CLIBRARY_H
2   #define CLIBRARY_H
3   typedef int (*callback_fcn)(int);
4   void some_c_func(callback_fcn);
5   #endif
```

Listing A.6: clibrary.c

```c
1   #include <stdio.h>
2
3   #include "clibrary.h"
4
5   void some_c_func(callback_fcn callback)
6   {
7           int arg = 2;
8           printf("C.some_c_func(): calling callback with arg
                = %d\n", arg);
9           int response = callback(2);
10          printf("C.some_c_func(): callback responded with %d
                \n", response);
11  }
```