



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Mastering Concurrency in Go

Discover and harness Go's powerful concurrency features to develop and build fast, scalable network systems

Nathan Kozyra

[PACKT] open source*
PUBLISHING community experience distilled

Mastering Concurrency in Go

Discover and harness Go's powerful concurrency features to develop and build fast, scalable network systems

Nathan Kozyra



Mastering Concurrency in Go

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: July 2014

Production reference: 1160714

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78398-348-3

www.packtpub.com

Cover image by Pratyush Mohanta (tysoncinematics@gmail.com)

Credits

Author

Nathan Kozyra

Project Coordinator

Danuta Jones

Reviewers

Jeremy R. Budnack

János Fehér

Aleksandar S. Sokolovski

Michele Della Torre

Proofreaders

Simran Bhogal

Ameesha Green

Paul Hindle

Commissioning Editor

Julian Ursell

Indexers

Hemangini Bari

Priya Subramani

Acquisition Editor

Kevin Colaco

Graphics

Abhinash Sahu

Content Development Editor

Athira Laji

Production Coordinator

Conidon Miranda

Technical Editors

Pratik More

Pooja Nair

Shweta S. Pant

Rohit Kumar Singh

Cover Work

Conidon Miranda

Copy Editors

Roshni Banerjee

Sarang Chari

Janbal Dharmaraj

Sayanee Mukherjee

Karuna Narayanan

Laxmi Subramanian

About the Author

Nathan Kozyra has been programming both recreationally and professionally for more than a decade now. His first experience came while writing games on his grandfather's Commodore 64, and in the ensuing decades, he's crafted technological solutions and applications in nearly every major language for a host of software and media companies as a developer, advisor, creative technologist, and CTO. He is currently the CTO of Pointslocal. A new-language enthusiast and C++ stalwart, his attention was quickly captured by Google's Go language, both for the language's creators and ethos as well as its apparent post-C approach to systems languages. Having dived in quickly, Go is now his go-to language for fast, concurrent applications.

This book is dedicated to Mary and Ethan.

About the Reviewers

Jeremy R. Budnack is a software engineer and consultant from Buffalo, NY. His career includes almost 15 years of experience in software development and systems administration, spanning many diverse technology stacks and programming languages (including C#, Ruby, and Go). He currently works for Stark & Wayne, LLC., specializing in building cloud-based solutions using a variety of technologies including Cloud Foundry, BOSH, OpenStack, and vSphere.

János Fehér has been involved in a wide variety of projects since 1996, including technical support for NATO operations and development of high-performance computing grids, national TV and radio websites, and web applications for universities and adult learning. In recent years, he has been heavily involved in distributed and concurrent software architectures. He is currently the Head of Development for Intern Avenue.

To my love, Szilvi, who painted my cycling shoes full with gophers.

Aleksandar Stefan Sokolovski is a project manager and an ICT research engineer from Skopje, Macedonia. He has a Bachelor of Science (BSc) degree in Computer Science from the Ss. Cyril and Methodius University and a Master of Science (MSc) degree in Technology, Innovation, and Entrepreneurship from the University of Sheffield, U.K. He has been a member of an organization committee (CITYR'09 / CITYR'11), a participant (CIIT'11), and a presenter (XIX Scientific Conference IT 2014) at multiple international research conferences; he is also a published author of many research papers for international conferences including a published paper in a journal (ETF Journal of Electrical Engineering, 2014). He has worked as a research associate and teaching assistant at the Faculty for Informatics and Computer Science in Skopje, Macedonia. He currently works as a research associate at the Institute for Digital Forensics and as an IT engineer, a research engineer, and a R&D project coordinator at telecommunications and Internet provider Neotel, Skopje from Republic of Macedonia. He is a member of the Executive Committee of IEEE R8, Section in Republic of Macedonia as Professional Activities Officer and is also a member of Project Management Institute (PMI)

I am heartily thankful to my mentors, professors Dragan Mihajlov, Toni, and Saso Gelev, whose encouragement, guidance, and support from the initial to the final level enabled me to develop an understanding and the knowledge to take on a significant part of this project. I offer my regards and respect to all my colleagues for their support and all their help. I would like to thank Packt Publishing and its team working on the book.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project: my family for their support and all their help, especially my sister Sofija, my parents Stefan and Brankica, and last but not least, my girlfriend Kalina.

Michele Della Torre is a software engineer, programmer, and geek born in Italy in 1982. In love with computers since he was a child, he started studying computer science at high school and obtained his Master's degree in Computer Engineering in 2007 at the Politecnico of Milan university. After university, he continued to learn about software design and architectures, agile methodologies, concurrent systems, and algorithms. He currently works in a home automation company and is occasionally involved in open source projects.

I'd like to thank my parents, my girlfriend Alessia, and my friends Alex and Fabio for their continuous support and patience.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: An Introduction to Concurrency in Go	9
Introducing goroutines	9
A patient goroutine	11
Implementing the defer control mechanism	13
Using Go's scheduler	15
Using system variables	20
Understanding goroutines versus coroutines	21
Implementing channels	22
Channel-based sorting at the letter capitalization factory	25
Cleaning up our goroutines	29
Buffered or unbuffered channels	29
Using the select statement	29
Closures and goroutines	33
Building a web spider using goroutines and channels	35
Summary	40
Chapter 2: Understanding the Concurrency Model	41
Understanding the working of goroutines	41
Synchronous versus asynchronous goroutines	42
Designing the web server plan	42
Visualizing concurrency	44
RSS in action	49
An RSS reader with self diagnostics	50
Imposing a timeout	57
A little bit about CSP	57
The dining philosophers problem	58
Go and the actor model	60

Object orientation	60
Demonstrating simple polymorphism in Go	61
Using concurrency	62
Managing threads	62
Using sync and mutexes to lock data	63
Summary	65
Chapter 3: Developing a Concurrent Strategy	67
Applying efficiency in complex concurrency	67
Identifying race conditions with race detection	68
Using mutual exclusions	72
Exploring timeouts	77
Importance of consistency	78
Synchronizing our concurrent operations	78
The project – multiuser appointment calendar	79
Visualizing a concurrent pattern	81
Developing our server requirements	82
Web server	82
Using templates	83
Time	84
Endpoints	84
Custom structs	85
A multiuser Appointments Calendar	85
A note on style	93
A note on immutability	94
Summary	95
Chapter 4: Data Integrity in an Application	97
Getting deeper with mutexes and sync	97
The cost of goroutines	100
Working with files	101
Getting low – implementing C	103
Touching memory in cgo	104
The structure of cgo	104
The other way around	105
Getting even lower – assembly in Go	109
Distributed Go	111
Some common consistency models	113
Distributed shared memory	113
First-in-first-out – PRAM	114
Looking at the master-slave model	114
The producer-consumer problem	115
Looking at the leader-follower model	116

Atomic consistency / mutual exclusion	117
Release consistency	117
Using memcached	118
Circuit	119
Summary	121
Chapter 5: Locks, Blocks, and Better Channels	123
Understanding blocking methods in Go	124
Blocking method 1 – a listening, waiting channel	124
Sending more data types via channels	125
The net package – a chat server with interfaced channels	130
Examining our client	137
Blocking method 2 – the select statement in a loop	139
Cleaning up goroutines	140
Blocking method 3 – network connections and reads	141
Creating channels of channels	141
Pprof – yet another awesome tool	143
Handling deadlocks and errors	150
Summary	151
Chapter 6: C10K – A Non-blocking Web Server in Go	153
Attacking the C10K problem	154
Failing of servers at 10,000 concurrent connections	155
Using concurrency to attack C10K	156
Taking another approach	158
Building our C10K web server	159
Benchmarking against a blocking web server	160
Handling requests	162
Routing requests	165
Serving pages	166
Parsing our template	167
External dependencies	171
Connecting to MySQL	171
Multithreading and leveraging multiple cores	174
Exploring our web server	174
Timing out and moving on	178
Summary	180
Chapter 7: Performance and Scalability	181
High performance in Go	182
Getting deeper into pprof	182
Parallelism's and concurrency's impact on I/O pprof	188
Using the App Engine	191

Distributed Go	192
Types of topologies	192
Type 1 – star	193
Type 2 – mesh	196
The Publish and Subscribe model	197
Serialized data	198
Remote code execution	198
Other topologies	199
Message Passing Interface	200
Some helpful libraries	201
Nitro profiler	201
Heka	202
GoFlow	202
Memory preservation	202
Garbage collection in Go	203
Summary	203
Chapter 8: Concurrent Application Architecture	205
Designing our concurrent application	206
Identifying our requirements	207
Using NoSQL as a data store in Go	208
MongoDB	209
Redis	211
Tiedot	213
CouchDB	214
Cassandra	215
Couchbase	216
Setting up our data store	219
Monitoring filesystem changes	221
Managing logfiles	222
Handling configuration files	223
Detecting file changes	224
Sending changes to clients	226
Checking records against Couchbase	229
Backing up our files	231
Designing our web interface	233
Reverting a file's history – command line	238
Using Go in daemons and as a service	240
Checking the health of our server	241
Summary	242

Chapter 9: Logging and Testing Concurrency in Go	243
Handling errors and logging	244
Breaking out goroutine logs	246
Using the LiteIDE for richer and easier debugging	248
Sending errors to screen	249
Logging errors to file	250
Logging errors to memory	252
Using the log4go package for robust logging	254
Panicking	259
Recovering	260
Logging our panics	263
Catching stack traces with concurrent code	265
Using the runtime package for granular stack traces	265
Summary	269
Chapter 10: Advanced Concurrency and Best Practices	271
Going beyond the basics with channels	272
Building workers	272
Implementing nil channel blocks	278
Using nil channels	280
Implementing more granular control over goroutines with tomb	281
Timing out with channels	284
Building a load balancer with concurrent patterns	286
Choosing unidirectional and bidirectional channels	292
Using receive-only or send-only channels	293
Using an indeterminate channel type	293
Using Go with unit testing	295
GoCheck	296
Ginkgo and Gomega	296
Using Google App Engine	297
Utilizing best practices	297
Structuring your code	298
Documenting your code	298
Making your code available via go get	299
Keeping concurrency out of your packages	299
Summary	300
Index	301

Preface

I just love new programming languages. Perhaps it's the inevitable familiarity and ennui with regard to existing languages and the frustration with existing tools, syntaxes, coding conventions, and performance. Maybe I'm just hunting for that one "language to rule them all". Whatever the reason, any time a new or experimental language is released, I have to dive right in.

This has been a golden age for new languages and language design. Think about it: the C language was released in the early 1970s — a time when resources were so scarce that verbosity, clarity, and syntactical logic were often eschewed for thrift. And most of the languages we use today were either originally written in this era or were directly influenced by those languages.

Since the late 1980s and early 1990s, there has been a slow flood of powerful new languages and paradigms — Perl, Python, Ruby, PHP, and JavaScript — have taken an expanding user base by storm and has become one of the most popular languages (up there with stalwarts such as C, C++, and Java). Multithreading, memory caching, and APIs have allowed multiple processes, dissonant languages, applications, and even separate operating systems to work in congress.

And while this is great, there's a niche that until very recently was largely unserved: powerful, compiled, cross-platform languages with concurrency support that are geared towards systems programmers.

Very few languages match these parameters. Sure, there have been lower-level languages that fulfill some of these characteristics. Erlang and Haskell fit the bill in terms of power and language design, but as functional languages they pose a learning barrier for systems programmers coming from a C/Java background. Objective-C and C# are relatively easy, powerful, and have concurrency support—but they're bound enough to a specific OS to make programming for other platforms arduous. The languages we just mentioned (Python, JavaScript, and so on)—while extremely popular—are largely interpreted languages, forcing performance into a secondary role. You can use most of them for systems programming, but in many ways it's the proverbial square peg in a round hole. So when Google announced Go in 2009, my interest was piqued. When I saw who was behind the project (more on that later), I was elated. When I saw the language and its design in action, I was in heaven.

For the last few years I've been using Go to replace systems applications I'd previously written in C, Java, Perl, and Python. I couldn't be happier with the results. Implementing Go has improved these applications in almost every instance. The fact that it plays nicely with C is another huge selling point for systems programmers looking to dip their toes in Go's pool.

With some of the best minds in language design (and programming in general) behind it, Go has a bright future.

For years—decades, really—there have been less than a handful of options for writing servers and network interfaces. If you were tasked with writing one, you probably reached for C, C++, or Java. And while these certainly can handle the task, and while they all now support concurrency and parallelism in some way or another, they weren't designed for that.

Google brought together a team that included some giants of programming—Rob Pike and Ken Thompson of Bell Labs fame and Robert Griesemer, who worked on Google's JavaScript implementation V8—to design a modern, concurrent language with development ease at the forefront.

To do this, the team focused on some sore spots in the alternatives, which are as follows:

- Dynamically typed languages have—in recent years—become incredibly popular. Go eschews the explicit, "cumbersome" type systems of Java or C++. Go uses type inference, which saves development time, but is still also strongly typed.
- Concurrency, parallelism, pointers/memory access, and garbage collection are unwieldy in the aforementioned languages. Go lets these concepts be as easy or as complicated as you want or need them to be.

- As a newer language, Go has a focus on multicore design that was a necessary afterthought in languages such as C++.
- Go's compiler is super-fast; it's so fast that there are implementations of it that treat Go code as interpreted.
- Although Google designed Go to be a systems language, it's versatile enough to be used in a myriad of ways. Certainly, the focus on advanced, cheap concurrency makes it ideal for network and systems programming.
- Go is loose with syntax, but strict with usage. By this we mean that Go will let you get a little lazy with some lexer tokens, but you still have to produce fundamentally tight code. As Go provides a formatting tool that attempts to clarify your code, you can also spend less time on readability concerns as you're coding.

What this book covers

Chapter 1, An Introduction to Concurrency in Go, introduces goroutines and channels, and will compare the way Go handles concurrency with the approach other languages use. We'll build some basic concurrent applications utilizing these new concepts.

Chapter 2, Understanding the Concurrency Model, focuses on resource allocation, sharing memory (and when not to), and data. We will look at channels and channels of channels as well as explain exactly how Go manages concurrency internally.

Chapter 3, Developing a Concurrent Strategy, discusses approach methods for designing applications to best use concurrent tools in Go. We'll look at some available third-party packages that can play a role in your strategy.

Chapter 4, Data Integrity in an Application, looks at ensuring that delegation of goroutines and channels maintain the state in single thread and multithread applications.

Chapter 5, Locks, Blocks, and Better Channels, looks at how Go can avoid dead locks out of the box, and when and where they can still occur despite Go's language design.

Chapter 6, C10K – A Non-blocking Web Server in Go, tackles one of the Internet's most famous and esteemed challenges and attempt to solve it with core Go packages. We'll then refine the product and test it against common benchmarking tools.

Chapter 7, Performance and Scalability, focuses on squeezing the most out of your concurrent Go code, best utilizing resources and accounting for and mitigating third-party software's impact on your own. We'll add some additional functionality to our web server and talk about other ways in which we can use these packages.

Chapter 8, Concurrent Application Architecture, focuses on when and where to implement concurrent patterns, when and how to utilize parallelism to take advantage of advanced hardware, and how to ensure data consistency.

Chapter 9, Logging and Testing Concurrency in Go, focuses on OS-specific methods for testing and deploying your application. We'll also look at Go's relationship with various code repositories.

Chapter 10, Advanced Concurrency and Best Practices, looks at more complicated and advanced techniques including duplicating concurrent features not available in Go's core.

What you need for this book

To work along with this book's examples, you'll need a computer running Windows, OS X, or quite a few Linux variants that support Go. For this book, our Linux examples and notes reference Ubuntu.

If you do not already have Go 1.3 or newer installed, you will need to get it either from the binaries download page on <http://golang.org/> or through your operating system's package manager.

To use all of the examples in this book, you'll also need to have the following software installed:

- MySQL (<http://dev.mysql.com/downloads/>)
- Couchbase (<http://www.couchbase.com/download>)

Your choice of IDE is a matter of personal preference, as anyone who's worked with developers can attest. That said, there are a few that lend themselves better to some languages than others and a couple that have good support for Go. This author uses Sublime Text, which plays very nice with Go, is lightweight, and allows you to build directly from within the IDE itself. Anywhere you see screenshots of code, it will be from within Sublime Text.

And while there's a good amount of baked-in support for Go code, there's also a nice plugin collection for Sublime Text called GoSublime, available at <https://github.com/DisposaBoy/GoSublime>.

Sublime Text isn't free, but there is a free evaluation version available that has no time limit. It's available in Windows, OS X, and Linux variants at <http://www.sublimetext.com/>.

Who this book is for

If you are a systems or network programmer with some knowledge of Go and concurrency, but would like to know about the implementation of concurrent systems written in Go this is the book for you. The goal of this book is to enable you to write high-performance, scalable, resource-thrifty systems and network applications in Go.

In this book, we'll write a number of basic and somewhat less - basic network and systems applications. It's assumed that you've worked with these types of applications before. If you haven't, some extracurricular study may be warranted to be able to fully digest this content.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `setProxy` function is called after every request, and you can see it as the first line in our handler."

A block of code is set as follows:

```
package main

import (
    "net/http"
    "html/template"
    "time"
    "regexp"
    "fmt"
    "io/ioutil"
    "database/sql"
    "log"
    "runtime"
    _ "github.com/go-sql-driver/mysql"
)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
package main

import (
    "fmt"
)

func stringReturn(text string) string {
    return text
}

func main() {
    myText := stringReturn("Here be the code")
    fmt.Println(myText)
}
```

Any command-line input or output is written as follows:

```
go get github.com/go-sql-driver/mysql
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "If you upload a file by dragging it to the **Drop files here to upload** box, within a few seconds you'll see that the file is noted as changed in the web interface."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

An Introduction to Concurrency in Go

While Go is both a great general purpose and low-level systems language, one of its primary strengths is the built-in concurrency model and tools. Many other languages have third-party libraries (or extensions), but inherent concurrency is something unique to modern languages, and it is a core feature of Go's design.

Though there's no doubt that Go excels at concurrency – as we'll see in this book – what it has that many other languages lack is a robust set of tools to test and build concurrent, parallel, and distributed code.

Enough talk about Go's marvelous concurrency features and tools, let's jump in.

Introducing goroutines

The primary method of handling concurrency is through a goroutine. Admittedly, our first piece of concurrent code (mentioned in the preface) didn't do a whole lot, simply spitting out alternating "hello"s and "world"s until the entire task was complete.

Here is that code once again:

```
package main

import (
    "fmt"
    "time"
)

type Job struct {
    i int
    max int
    text string
}

func outputText(j *Job) {
    for j.i < j.max {
        time.Sleep(1 * time.Millisecond)
        fmt.Println(j.text)
        j.i++
    }
}

func main() {
    hello := new(Job)
    world := new(Job)

    hello.text = "hello"
    hello.i = 0
    hello.max = 3

    world.text = "world"
    world.i = 0
    world.max = 5

    go outputText(hello)
    outputText(world)
}
```



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

But, if you think back to our real-world example of planning a surprise party for your grandmother, that's exactly how things often have to be managed with limited or finite resources. This asynchronous behavior is critical for some applications to run smoothly, although our example essentially ran in a vacuum.

You may have noticed one quirk in our early example: despite the fact that we called the `outputText()` function on the `hello` struct first, our output started with the `world` struct's text value. Why is that?

Being asynchronous, when a goroutine is invoked, it waits for the blocking code to complete before concurrency begins. You can test this by replacing the `outputText()` function call on the `world` struct with a goroutine, as shown in the following code:

```
go outputText(hello)
go outputText(world)
```

If you run this, you will get no output because the main function ends while the asynchronous goroutines are running. There are a couple of ways to stop this to see the output before the main function finishes execution and the program exits. The classic method simply asks for user input before execution, allowing you to directly control when the application finishes. You can also put an infinite loop at the end of your main function, as follows:

```
for {}
```

Better yet, Go also has a built-in mechanism for this, which is the `WaitGroup` type in the `sync` package.

If you add a `WaitGroup` struct to your code, it can delay execution of the main function until after all goroutines are complete. In simple terms, it lets you set a number of required iterations to get a completed response from the goroutines before allowing the application to continue. Let's look at a minor modification to our "Hello World" application in the following section.

A patient goroutine

From here, we'll implement a `WaitGroup` struct to ensure our goroutines run entirely before moving on with our application. In this case, when we say patient, it's in contrast to the way we've seen goroutines run outside of a parent method with our previous example. In the following code, we will implement our first `waitgroup` struct:

```
package main

import (
    "fmt"
    "sync"
    "time"
)
```

```
type Job struct {
    i int
    max int
    text string
}

func outputText(j *Job, goGroup *sync.WaitGroup) {
    for j.i < j.max {
        time.Sleep(1 * time.Millisecond)
        fmt.Println(j.text)
        j.i++
    }
    goGroup.Done()
}

func main() {

    goGroup := new(sync.WaitGroup)
    fmt.Println("Starting")

    hello := new(Job)
    hello.text = "hello"
    hello.i = 0
    hello.max = 2

    world := new(Job)
    world.text = "world"
    world.i = 0
    world.max = 2

    go outputText(hello, goGroup)
    go outputText(world, goGroup)

    goGroup.Add(2)
    goGroup.Wait()

}
```

Let's look at the changes in the following code:

```
goGroup := new(sync.WaitGroup)
```

Here, we declared a `WaitGroup` struct named `goGroup`. This variable will receive note that our goroutine function has completed x number of times before allowing the program to exit. Here's an example of sending such an expectation in `WaitGroup`:

```
goGroup.Add(2)
```

The `Add()` method specifies how many `Done` messages `goGroup` should receive before satisfying its wait. Here, we specified 2 because we have two functions running asynchronously. If you had three goroutine members and still called two, you may see the output of the third. If you added a value more than two to `goGroup`, for example, `goGroup.Add(3)`, then `WaitGroup` would wait forever and deadlock.

With that in mind, you shouldn't manually set the number of goroutines that need to wait; this is ideally handled computationally or explicitly in a range. This is how we tell `WaitGroup` to wait:

```
goGroup.Wait()
```

Now, we wait. This code will fail for the same reason `goGroup.Add(3)` failed; the `goGroup` struct never receives messages that our goroutines are done. So, let's do this as shown in the following code snippet:

```
func outputText(j *Job, goGroup *sync.WaitGroup) {
    for j.i < j.max {
        time.Sleep(1 * time.Millisecond)
        fmt.Println(j.text)
        j.i++
    }
    goGroup.Done()
}
```

We've only made two changes to our `outputText()` function from the preface. First, we added a pointer to our `goGroup` as the second function argument. Then, when all our iterations were complete, we told `goGroup` that they are all done.

Implementing the defer control mechanism

While we're here, we should take a moment and talk about `defer`. Go has an elegant implementation of the defer control mechanism. If you've used `defer` (or something functionally similar) in other languages, this will seem familiar—it's a useful way of delaying the execution of a statement until the rest of the function is complete.

For the most part, this is syntactical sugar that allows you to see related operations together, even though they won't execute together. If you've ever written something similar to the following pseudocode, you'll know what I mean:

```
x = file.open('test.txt')
int longFunction() {
...
}
x.close();
```

You probably know the kind of pain that can come from large "distances" separating related bits of code. In Go, you can actually write the code similar to the following:

```
package main

import(
    "os"
)

func main() {

    file, _ := os.Create("/defer.txt")

    defer file.Close()

    for {

        break

    }

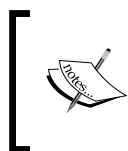
}
```

There isn't any actual functional advantage to this other than making clearer, more readable code, but that's a pretty big plus in itself. Deferred calls are executed reverse of the order in which they are defined, or last-in-first-out. You should also take note that any data passed by reference may be in an unexpected state.

For example, refer to the following code snippet:

```
func main() {  
  
    aValue := new(int)  
  
    defer fmt.Println(*aValue)  
  
    for i := 0; i < 100; i++ {  
        *aValue++  
    }  
  
}
```

This will return 0, and not 100, as it is the default value for an integer.



Defer is not the same as *deferred* (or futures/promises) in other languages. We'll talk about Go's implementations and alternatives to futures and promises in *Chapter 2, Understanding the Concurrency Model*.

Using Go's scheduler

With a lot of concurrent and parallel applications in other languages, the management of both soft and hard threads is handled at the operating system level. This is known to be inherently inefficient and expensive as the OS is responsible for context switching, among multiple processes. When an application or process can manage its own threads and scheduling, it results in faster runtime. The threads granted to our application and Go's scheduler have fewer OS attributes that need to be considered in context to switching, resulting in less overhead.

If you think about it, this is self-evident—the more you have to juggle, the slower it is to manage all of the balls. Go removes the natural inefficiency of this mechanism by using its own scheduler.

There's really only one quirk to this, one that you'll learn very early on: if you don't ever yield to the main thread, your goroutines will perform in unexpected ways (or won't perform at all).

Another way to look at this is to think that a goroutine must be blocked before concurrency is valid and can begin. Let's modify our example and include some file I/O to log to demonstrate this quirk, as shown in the following code:

```
package main

import (
    "fmt"
    "time"
    "io/ioutil"
)

type Job struct {
    i int
    max int
    text string
}

func outputText(j *Job) {
    fileName := j.text + ".txt"
    fileContents := ""
    for j.i < j.max {
        time.Sleep(1 * time.Millisecond)
        fileContents += j.text
        fmt.Println(j.text)
        j.i++
    }
    err := ioutil.WriteFile(fileName, []byte(fileContents), 0644)
    if (err != nil) {
        panic("Something went awry")
    }
}

func main() {

    hello := new(Job)
    hello.text = "hello"
    hello.i = 0
    hello.max = 3
}
```

```
world := new(Job)
world.text = "world"
world.i = 0
world.max = 5

go outputText(hello)
go outputText(world)

}
```

In theory, all that has changed is that we're now using a file operation to log each operation to a distinct file (in this case, `hello.txt` and `world.txt`). However, if you run this, no files are created.

In our last example, we used a `sync.WaitSync` struct to force the main thread to delay execution until asynchronous tasks were complete. While this works (and elegantly), it doesn't really explain *why* our asynchronous tasks fail. As mentioned before, you can also utilize blocking code to prevent the main thread from completing before its asynchronous tasks.

Since the Go scheduler manages context switching, each goroutine must yield control back to the main thread to schedule all of these asynchronous tasks. There are two ways to do this manually. One method, and probably the ideal one, is the `WaitGroup` struct. Another is the `GoSched()` function in the `runtime` package.

The `GoSched()` function temporarily yields the processor and then returns to the current goroutine. Consider the following code as an example:

```
package main

import (
    "runtime"
    "fmt"
)

func showNumber(num int) {
    fmt.Println(num)
}

func main() {
    iterations := 10

    for i := 0; i<=iterations; i++ {
```



```
        go showNumber(i)

    }
    //runtime.Gosched()
    fmt.Println("Goodbye!")
}
```

Run this with `runtime.Gosched()` commented out and the underscore before `"runtime"` removed, and you'll see only `Goodbye!`. This is because there's no guarantee as to how many goroutines, if any, will complete before the end of the `main()` function.

As we learned earlier, you can explicitly wait for a finite set number of goroutines before ending the execution of the application. However, `Gosched()` allows (in most cases) for the same basic functionality. Remove the comment before `runtime.Gosched()`, and you should get 0 through 10 printed before `Goodbye!`.

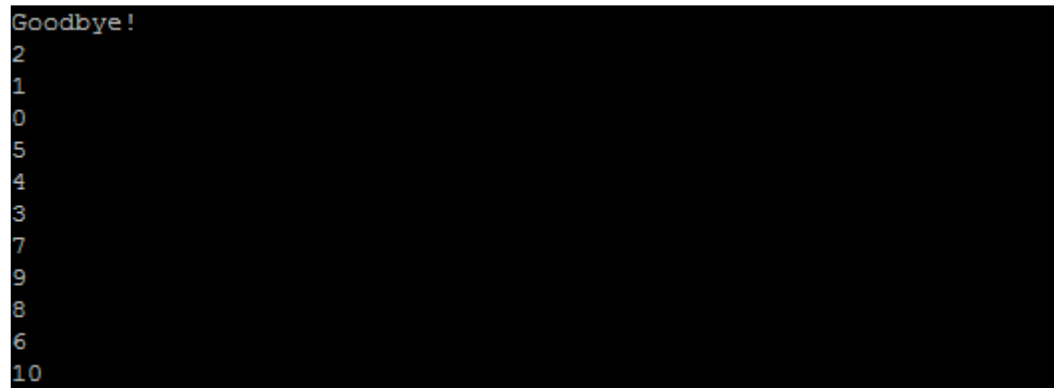
Just for fun, try running this code on a multicore server and modify your max processors using `runtime.GOMAXPROCS()`, as follows:

```
func main() {

    runtime.GOMAXPROCS(2)
```

Also, push your `runtime.Gosched()` to the absolute end so that all goroutines have a chance to run before `main` ends.

Got something unexpected? That's not unexpected! You may end up with a totally jostled execution of your goroutines, as shown in the following screenshot:



```
Goodbye!
2
1
0
5
4
3
7
9
8
6
10
```

Although it's not entirely necessary to demonstrate how juggling your goroutines with multiple cores can be vexing, this is one of the simplest ways to show exactly why it's important to have communication between them (and the Go scheduler).

You can debug the parallelism of this using `GOMAXPROCS > 1`, enveloping your goroutine call with a timestamp display, as follows:

```
tstamp := strconv.FormatInt(time.Now().UnixNano(), 10)
fmt.Println(num, tstamp)
```



Remember to import the `time` and `strconv` parent packages here.

This will also be a good place to see concurrency and compare it to parallelism in action. First, add a one-second delay to the `showNumber()` function, as shown in the following code snippet:

```
func showNumber(num int) {
    tstamp := strconv.FormatInt(time.Now().UnixNano(), 10)
    fmt.Println(num, tstamp)
    time.Sleep(time.Millisecond * 10)
}
```

Then, remove the goroutine call before the `showNumber()` function with `GOMAXPROCS(0)`, as shown in the following code snippet:

```
runtime.GOMAXPROCS(0)
iterations := 10

for i := 0; i<=iterations; i++ {
    showNumber(i)
}
```

As expected, you get 0-10 with 10-millisecond delays between them followed by Goodbye! as an output. This is straight, serial computing.

Next, let's keep `GOMAXPROCS` at zero for a single thread, but restore the goroutine as follows:

```
go showNumber(i)
```

This is the same process as before, except for the fact that everything will execute within the same general timeframe, demonstrating the concurrent nature of execution. Now, go ahead and change your `GOMAXPROCS` to two and run again. As mentioned earlier, there is only one (or possibly two) timestamp, but the order has changed because everything is running simultaneously.

Goroutines aren't (necessarily) thread-based, but they feel like they are. When Go code is compiled, the goroutines are multiplexed across available threads. It's this very reason why Go's scheduler needs to know what's running, what needs to finish before the application's life ends, and so on. If the code has two threads to work with, that's what it will use.

Using system variables

So what if you want to know how many threads your code has made available to you?

Go has an environment variable returned from the runtime package function `GOMAXPROCS`. To find out what's available, you can write a quick application similar to the following code:

```
package main

import (
    "fmt"
    "runtime"
)

func listThreads() int {
    threads := runtime.GOMAXPROCS(0)
    return threads
}

func main() {
    runtime.GOMAXPROCS(2)
    fmt.Printf("%d thread(s) available to Go.", listThreads())
}
```

A simple Go build on this will yield the following output:

```
2 thread(s) available to Go.
```

The `0` parameter (or no parameter) delivered to `GOMAXPROCS` means no change is made. You can put another number in there, but as you might imagine, it will only return what is actually available to Go. You cannot exceed the available cores, but you can limit your application to use less than what's available.

The `GOMAXPROCS()` call itself returns an integer that represents the *previous* number of processors available. In this case, we first set it to two and then set it to zero (no change), returning two.

It's also worth noting that increasing `GOMAXPROCS` can sometimes *decrease* the performance of your application.

As there are context-switching penalties in larger applications and operating systems, increasing the number of threads used means goroutines can be shared among more than one, and the lightweight advantage of goroutines might be sacrificed.

If you have a multicore system, you can test this pretty easily with Go's internal benchmarking functionality. We'll take a closer look at this functionality in *Chapter 5, Locks, Blocks, and Better Channels*, and *Chapter 7, Performance and Scalability*.

The runtime package has a few other very useful environment variable return functions, such as `NumCPU`, `NumGoroutine`, `CPUProfile`, and `BlockProfile`. These aren't just handy to debug, they're also good to know how to best utilize your resources. This package also plays well with the `reflect` package, which deals with metaprogramming and program self-analysis. We'll touch on that in more detail later in *Chapter 9, Logging and Testing Concurrency in Go*, and *Chapter 10, Advanced Concurrency and Best Practices*.

Understanding goroutines versus coroutines

At this point, you may be thinking, "Ah, goroutines, I know these as coroutines." Well, yes and no.

A coroutine is a cooperative task control mechanism, but in its most simplistic sense, a coroutine is not concurrent. While coroutines and goroutines are utilized in similar ways, Go's focus on concurrency provides a lot more than just state control and yields. In the examples we've seen so far, we have what we can call *dumb* goroutines. Although they operate in the same time and address space, there's no real communication between the two. If you look at coroutines in other languages, you may find that they are often not necessarily concurrent or asynchronous, but rather they are step-based. They yield to `main()` and to each other, but two coroutines might not necessarily communicate between each other, relying on a centralized, explicitly written data management system.

The original coroutine

Coroutines were first described for COBOL by Melvin Conway. In his paper, *Design of a Separable Transition-Diagram Compiler*, he suggested that the purpose of a coroutine was to take a program broken apart into subtasks and allow them to operate independently, sharing only small pieces of data.



Coroutines can sometimes violate the basic tenets of Conway's coroutines. For example, Conway suggested that there should be only a unidirectional path of execution; in other words, A followed by B, then C, and then D, and so on, where each represents an application chunk in a coroutine. We know that goroutines can be run in parallel and can execute in a seemingly arbitrary order (at least without direction). To this point, our goroutines have not shared any information either; they've simply executed in a shared pattern.

Implementing channels

So far, we've dabbled in concurrent processes that are capable of doing a lot but not effectively communicating with each other. In other words, if you have two processes occupying the same processing time and sharing the same memory and data, you must have a way of knowing which process is in which place as part of a larger task.

Take, for example, an application that must loop through one paragraph of Lorem Ipsum and capitalize each letter, then write the result to a file. Of course, we will not really need a concurrent application to do this (and in fact, it's an endemic function of almost any language that handles strings), but it's a quick way to demonstrate the potential limitations of isolated goroutines. Shortly, we'll turn this primitive example into something more practical, but for now, here's the beginning of our capitalization example:

```
package main

import (
    "fmt"
    "runtime"
    "strings"
)
```

```
var loremIpsum string
var finalIpsum string
var letterSentChan chan string

func deliverToFinal(letter string, finalIpsum *string) {
    *finalIpsum += letter
}

func capitalize(current *int, length int, letters []byte,
    finalIpsum *string) {
    for *current < length {
        thisLetter := strings.ToUpper(string(letters[*current]))

        deliverToFinal(thisLetter, finalIpsum)
        *current++
    }
}

func main() {

    runtime.GOMAXPROCS(2)

    index := new(int)
    *index = 0
    loremIpsum = "Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Vestibulum venenatis magna eget libero tincidunt, ac
condimentum enim auctor. Integer mauris arcu, dignissim sit amet
convallis vitae, ornare vel odio. Phasellus in lectus risus. Ut
sodales vehicula ligula eu ultricies. Fusce vulputate fringilla
eros at congue. Nulla tempor neque enim, non malesuada arcu
laoreet quis. Aliquam eget magna metus. Vivamus lacinia
venenatis dolor, blandit faucibus mi iaculis quis. Vestibulum
sit amet feugiat ante, eu porta justo."

    letters := []byte(loremIpsum)
    length := len(letters)

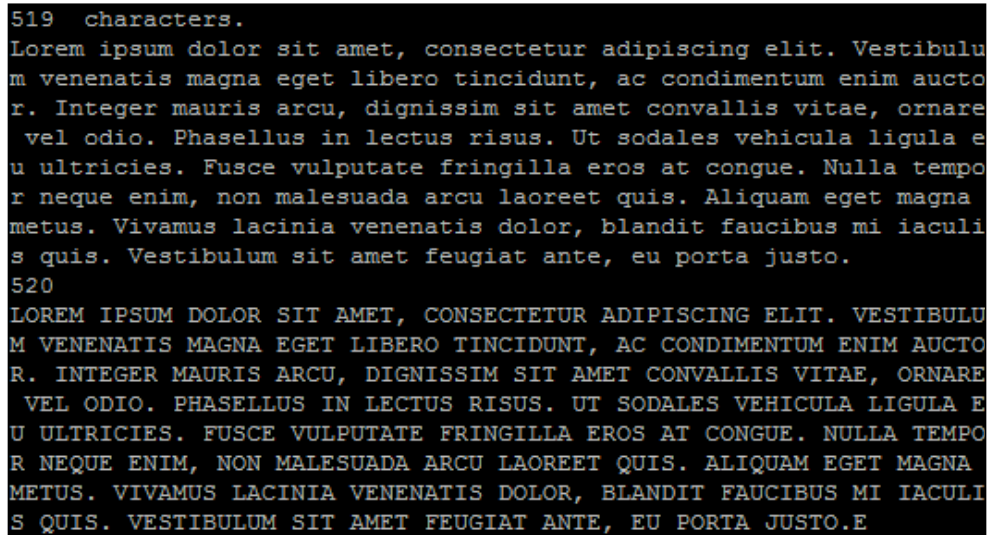
    go capitalize(index, length, letters, &finalIpsum)
```

```
go func() {
    go capitalize(index, length, letters, &finalIpsum)
}()

fmt.Println(length, " characters.")
fmt.Println(loremIpsum)
fmt.Println(*index)
fmt.Println(finalIpsum)

}
```

If we run this with some degree of parallelism here but no communication between our goroutines, we'll end up with a jumbled mess of text, as shown in the following screenshot:



```
519  characters.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulu
m venenatis magna eget libero tincidunt, ac condimentum enim aucto
r. Integer mauris arcu, dignissim sit amet convallis vitae, ornare
vel odio. Phasellus in lectus risus. Ut sodales vehicula ligula e
u ultricies. Fusce vulputate fringilla eros at congue. Nulla tempo
r neque enim, non malesuada arcu laoreet quis. Aliquam eget magna
metus. Vivamus lacinia venenatis dolor, blandit faucibus mi iaculi
s quis. Vestibulum sit amet feugiat ante, eu porta justo.
520
LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT. VESTIBULU
M VENENATIS MAGNA EGET LIBERO TINCIDUNT, AC CONDIMENTUM ENIM AUCTO
R. INTEGER MAURIS ARCU, DIGNISSIM SIT AMET CONVALLIS VITAE, ORNARE
VEL ODIO. PHASELLUS IN LECTUS RISUS. UT SODALES VEHICULA LIGULA E
U ULTRICIES. FUSCE VULPUTATE FRINGILLA EROS AT CONGUE. NULLA TEMPO
R NEQUE ENIM, NON MALESUADA ARCU LAOREET QUIS. ALIQUAM EGET MAGNA
METUS. VIVAMUS LACINIA VENENATIS DOLOR, BLANDIT FAUCIBUS MI IACULI
S QUIS. VESTIBULUM SIT AMET FEUGIAT ANTE, EU PORTA JUSTO.E
```

Due to the demonstrated unpredictability of concurrent scheduling in Go, it may take many iterations to get this exact output. In fact, you may never get the exact output.

This won't do, obviously. So how do we best structure this application? The missing piece here is synchronization, but we could also do with a better design pattern.

Here's another way to break this problem down into pieces. Instead of having two processes handling the same thing in parallel, which is rife with risk, let's have one process that takes a letter from the `loremIpsum` string and capitalizes it, and then pass it onto another process to add it to our `finalIpsum` string.

You can envision this as two people sitting at two desks, each with a stack of letters. Person A is responsible to take a letter and capitalize it. He then passes the letter onto person B, who then adds it to the `finalIpsum` stack. To do this, we'll implement a channel in our code in an application tasked with taking text (in this case, the first line of Abraham Lincoln's Gettysburg address) and capitalizing each letter.

Channel-based sorting at the letter capitalization factory

Let's take the last example and do something (slightly) more purposeful by attempting to capitalize the preamble of Abraham Lincoln's Gettysburg address while mitigating the sometimes unpredictable effect of concurrency in Go, as shown in the following code:

```
package main

import (
    "fmt"
    "sync"
    "runtime"
    "strings"
)

var initialString string
var finalString string

var stringLength int

func addToFinalStack(letterChannel chan string, wg
    *sync.WaitGroup) {
    letter := <-letterChannel
    finalString += letter
    wg.Done()
}

func capitalize(letterChannel chan string, currentLetter string,
    wg *sync.WaitGroup) {

    thisLetter := strings.ToUpper(currentLetter)
```



```
    wg.Done()
    letterChannel <- thisLetter
}

func main() {

    runtime.GOMAXPROCS(2)
    var wg sync.WaitGroup

    initialString = "Four score and seven years ago our fathers
    brought forth on this continent, a new nation, conceived in
    Liberty, and dedicated to the proposition that all men are
    created equal."
    initialBytes := []byte(initialString)

    var letterChannel chan string = make(chan string)

    stringLength = len(initialBytes)

    for i := 0; i < stringLength; i++ {
        wg.Add(2)

        go capitalize(letterChannel, string(initialBytes[i]), &wg)
        go addToFinalStack(letterChannel, &wg)

        wg.Wait()
    }

    fmt.Println(finalString)
}
```

You'll note that we even bumped this up to a duo-core process and ended up with the following output:

```
go run alpha-channel.go
```

```
FOUR SCORE AND SEVEN YEARS AGO OUR FATHERS BROUGHT FORTH ON THIS
CONTINENT, A NEW NATION, CONCEIVED IN LIBERTY, AND DEDICATED TO THE
PROPOSITION THAT ALL MEN ARE CREATED EQUAL.
```

The output is just as we expected. It's worth reiterating that this example is overkill of the most extreme kind, but we'll parlay this functionality into a usable practical application shortly.

So what's happening here? First, we reimplemented the `sync.WaitGroup` struct to allow all of our concurrent code to execute while keeping the main thread alive, as shown in the following code snippet:

```
var wg sync.WaitGroup
...
for i := 0; i < stringLength; i++ {
    wg.Add(2)

    go capitalize(letterChannel, string(initialBytes[i]), &wg)
    go addToFinalStack(letterChannel, &wg)

    wg.Wait()
}
```

We allow each goroutine to tell the `WaitGroup` struct that we're done with the step. As we have two goroutines, we queue two `Add()` methods to the `WaitGroup` struct. Each goroutine is responsible to announce that it's done.

Next, we created our first channel. We instantiate a channel with the following line of code:

```
var letterChannel chan string = make(chan string)
```

This tells Go that we have a channel that will send and receive a string to various procedures/goroutines. This is essentially the manager of all of the goroutines. It is also responsible to send and receive data to goroutines and manage the order of execution. As we mentioned earlier, the ability of channels to operate with internal context switching and without reliance on multithreading permits them to operate very quickly.

There is a built-in limit to this functionality. If you design non-concurrent or blocking code, you will effectively remove concurrency from goroutines. We will talk more about this shortly.

We run two separate goroutines through `letterChannel`: `capitalize()` and `addToFinalStack()`. The first one simply takes a single byte from a byte array constructed from our string and capitalizes it. It then returns the byte to the channel as shown in the following line of code:

```
letterChannel <- thisLetter
```

All communication across a channel happens in this fashion. The `<-` symbol syntactically tells us that data will be sent back to (or back through) a channel. It's never necessary to do anything with this data, but the most important thing to know is that a channel can be blocking, at least per thread, until it receives data back. You can test this by creating a channel and then doing absolutely nothing of value with it, as shown in the following code snippet:

```
package main

func doNothing() (string) {

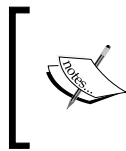
    return "nothing"
}

func main() {

    var channel chan string = make(chan string)
    channel <- doNothing()

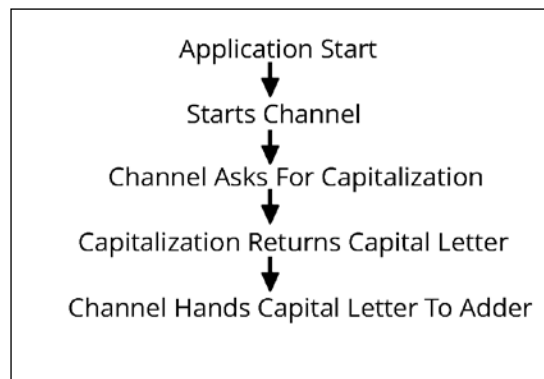
}
```

As nothing is sent along the channel and no goroutine is instantiated, this results in a deadlock. You can fix this easily by creating a goroutine and by bringing the channel into the global space by creating it outside of `main()`.



For the sake of clarity, our example here uses a local scope channel. Keeping these global whenever possible removes a lot of cruft, particularly if you have a lot of goroutines, as references to the channel can clutter up your code in a hurry.

For our example as a whole, you can look at it as is shown in the following figure:



Cleaning up our goroutines

You may be wondering why we need a `WaitGroup` struct when using channels. After all, didn't we say that a channel gets blocked until it receives data? This is true, but it requires one other piece of syntax.

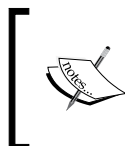
A nil or uninitialized channel will always get blocked. We will discuss the potential uses and pitfalls of this in *Chapter 7, Performance and Scalability*, and *Chapter 10, Advanced Concurrency and Best Practices*.

You have the ability to dictate how a channel blocks the application based on a second option to the `make` command by dictating the channel buffer.

Buffered or unbuffered channels

By default, channels are unbuffered, which means they will accept anything sent on them if there is a channel ready to receive. It also means that every channel call will block the execution of the application. By providing a buffer, the channel will only block the application when many returns have been sent.

A buffered channel is synchronous. To guarantee asynchronous performance, you'll want to experiment by providing a buffer length. We'll look at ways to ensure our execution falls as we expect in the next chapter.



Go's channel system is based on **Communicating Sequential Processes (CSP)**, a formal language to design concurrent patterns and multiprocessing. You will likely encounter CSP on its own when people describe goroutines and channels.

Using the select statement

One of the issues with first implementing channels is that whereas goroutines were formerly the method of simplistic and concurrent execution of code, we now have a single-purpose channel that dictates application logic across the goroutines. Sure, the channel is the traffic manager, but it never knows when traffic is coming, when it's no longer coming, and when to go home, unless being explicitly told. It waits passively for communication and can cause problems if it never receives any.

Go has a select control mechanism, which works just as effectively as a `switch` statement does, but on channel communication instead of variable values. A `switch` statement modifies execution based on the value of a variable, and `select` reacts to actions and communication across a channel. You can use this to orchestrate and arrange the control flow of your application. The following code snippet is our traditional `switch`, familiar to Go users and common among other languages:

```
switch {  
  
    case 'x':  
  
    case 'y':  
  
}
```

The following code snippet represents the `select` statement:

```
select {  
  
    case <- channelA:  
  
    case <- channelB:  
  
}
```

In a `switch` statement, the right-hand expression represents a value; in `select`, it represents a receive operation on a channel. A `select` statement will block the application until some information is sent along the channel. If nothing is sent ever, the application deadlocks and you'll get an error to that effect.

If two receive operations are sent at the same time (or if two cases are otherwise met), Go will evaluate them in an unpredictable fashion.

So, how might this be useful? Let's look at a modified version of the letter capitalization application's main function:

```
package main  
  
import(  
    "fmt"  
    "strings"  
)
```

```
var initialString string
var initialBytes []byte
var stringLength int
var finalString string
var lettersProcessed int
var applicationStatus bool
var wg sync.WaitGroup

func getLetters(gQ chan string) {

    for i := range initialBytes {
        gQ <- string(initialBytes[i])
    }
}

func capitalizeLetters(gQ chan string, sQ chan string) {

    for {
        if lettersProcessed >= stringLength {
            applicationStatus = false
            break
        }
        select {
            case letter := <- gQ:
                capitalLetter := strings.ToUpper(letter)
                finalString += capitalLetter
                lettersProcessed++
        }
    }
}

func main() {

    applicationStatus = true;

    getQueue := make(chan string)
    stackQueue := make(chan string)
```

```
    initialString = "Four score and seven years ago our fathers  
    brought forth on this continent, a new nation, conceived in  
    Liberty, and dedicated to the proposition that all men are  
    created equal."  
    initialBytes = []byte(initialString)  
    stringLength = len(initialString)  
    lettersProcessed = 0  
  
    fmt.Println("Let's start capitalizing")  
  
    go getLetters(getQueue)  
    capitalizeLetters(getQueue, stackQueue)  
  
    close(getQueue)  
    close(stackQueue)  
  
    for {  
        if applicationStatus == false {  
            fmt.Println("Done")  
            fmt.Println(finalString)  
            break  
        }  
    }  
}
```

The primary difference here is we now have a channel that listens for data across two functions running concurrently, `getLetters` and `capitalizeLetters`. At the bottom, you'll see a `for{}` loop that keeps the main active until the `applicationStatus` variable is set to false. In the following code, we pass each of these bytes as a string through the Go channel:

```
func getLetters(gQ chan string) {  
    for i := range initialBytes {  
        gQ <- string(initialBytes[i])  
    }  
}
```

The `getLetters` function is our primary goroutine that fetches individual letters from the byte array constructed from Lincoln's line. As the function iterates through each byte, it sends the letter through the `getQueue` channel.

On the receiving end, we have `capitalizeLetters` that takes each letter as it's sent across the channel, capitalizes it, and appends to our `finalString` variable. Let's take a look at this:

```
func capitalizeLetters(gQ chan string, sQ chan string) {  
  
    for {  
        if lettersProcessed >= stringLength {  
            applicationStatus = false  
            break  
        }  
        select {  
            case letter := <- gQ:  
                capitalLetter := strings.ToUpper(letter)  
                finalString += capitalLetter  
                lettersProcessed++  
        }  
    }  
}
```

It's critical that all channels are closed at some point or our application will hit a deadlock. If we never break the `for` loop here, our channel will be left waiting to receive from a concurrent process, and the program will deadlock. We manually check to see that we've capitalized all letters and only then break the loop.

Closures and goroutines

You may have noticed the anonymous goroutine in Lorem Ipsum:

```
go func() {  
    go capitalize(index, length, letters, &finalIpsum)  
}()
```

While it isn't always ideal, there are plenty of places where inline functions work best in creating a goroutine.

The easiest way to describe this is to say that a function isn't big or important enough to deserve a named function, but the truth is, it's more about readability. If you have dealt with lambdas in other languages, this probably doesn't need much explanation, but try to reserve these for quick inline functions.

In the earlier examples, the closure works largely as a wrapper to invoke a `select` statement or to create anonymous goroutines that will feed the `select` statement.

Since functions are first-class citizens in Go, not only can you utilize inline or anonymous functions directly in your code, but you can also pass them to and from other functions.

Here's an example that passes a function's result as a return value, keeping the state resolute outside of that returned function. In this, we'll return a function as a variable and iterate initial values on the returned function. The initial argument will accept a string that will be trimmed by word length with each successive call of the returned function.

```
import (
    "fmt"
    "strings"
)

func shortenString(message string) func() string {

    return func() string {
        messageSlice := strings.Split(message, " ")
        wordLength := len(messageSlice)
        if wordLength < 1 {
            return "Nothingn Left!"
        }else {
            messageSlice = messageSlice[:wordLength-1]
            message = strings.Join(messageSlice, " ")
            return message
        }
    }
}

func main() {

    myString := shortenString("Welcome to concurrency in Go! ...")
```

```
    fmt.Println(myString())  
    fmt.Println(myString())  
    fmt.Println(myString())  
    fmt.Println(myString())  
    fmt.Println(myString())  
    fmt.Println(myString())  
}
```

Once initialized and returned, we set the message variable, and each successive run of the returned method iterates on that value. This functionality allows us to eschew running a function multiple times on returned values or loop unnecessarily when we can very cleanly handle this with a closure as shown.

Building a web spider using goroutines and channels

Let's take the largely useless capitalization application and do something practical with it. Here, our goal is to build a rudimentary spider. In doing so, we'll accomplish the following tasks:

- Read five URLs
- Read those URLs and save the contents to a string
- Write that string to a file when all URLs have been scanned and read

These kinds of applications are written every day, and they're the ones that benefit the most from concurrency and non-blocking code.

It probably goes without saying, but this is not a particularly elegant web scraper. For starters, it only knows a few start points – the five URLs that we supply it. Also, it's neither recursive nor is it thread-safe in terms of data integrity.

That said, the following code works and demonstrates how we can use channels and the `select` statements:

```
package main  
  
import (  
    "fmt"  
    "io/ioutil"  
    "net/http"  
    "time"  
)
```

```
var applicationStatus bool
var urls []string
var urlsProcessed int
var foundUrls []string
var fullText string
var totalURLCount int
var wg sync.WaitGroup

var v1 int
```

First, we have our most basic global variables that we'll use for the application state. The `applicationStatus` variable tells us that our spider process has begun and `urls` is our slice of simple string URLs. The rest are idiomatic data storage variables and/or application flow mechanisms. The following code snippet is our function to read the URLs and pass them across the channel:

```
func readURLs(statusChannel chan int, textChannel chan string) {

    time.Sleep(time.Millisecond * 1)
    fmt.Println("Grabbing", len(urls), "urls")
    for i := 0; i < totalURLCount; i++ {

        fmt.Println("Url", i, urls[i])
        resp, _ := http.Get(urls[i])
        text, err := ioutil.ReadAll(resp.Body)

        textChannel <- string(text)

        if err != nil {
            fmt.Println("No HTML body")
        }

        statusChannel <- 0

    }

}
```

The `readURLs` function assumes `statusChannel` and `textChannel` for communication and loops through the `urls` variable slice, returning the text on `textChannel` and a simple ping on `statusChannel`. Next, let's look at the function that will append scraped text to the full text:

```
func addToScrapedText(textChannel chan string, processChannel chan
    bool) {
```

```
for {
    select {
    case pC := <-processChannel:
        if pC == true {
            // hang on
        }
        if pC == false {

            close(textChannel)
            close(processChannel)
        }
    case tC := <-textChannel:
        fullText += tC
    }
}

}
```

We use the `addToScrapedText` function to accumulate processed text and add it to a master text string. We also close our two primary channels when we get a kill signal on our `processChannel`. Let's take a look at the `evaluateStatus()` function:

```
func evaluateStatus(statusChannel chan int, textChannel chan
    string, processChannel chan bool) {

    for {
        select {
        case status := <-statusChannel:

            fmt.Print(urlsProcessed, totalURLCount)
            urlsProcessed++
            if status == 0 {

                fmt.Println("Got url")
            }
            if status == 1 {
```

```
        close(statusChannel)
    }
    if urlsProcessed == totalURLCount {
        fmt.Println("Read all top-level URLs")
        processChannel <- false
        applicationStatus = false
    }
}
}
```

At this juncture, all that the `evaluateStatus` function does is determine what's happening in the overall scope of the application. When we send a 0 (our aforementioned ping) through this channel, we increment our `urlsProcessed` variable. When we send a 1, it's a message that we can close the channel. Finally, let's look at the main function:

```
func main() {
    applicationStatus = true
    statusChannel := make(chan int)
    textChannel := make(chan string)
    processChannel := make(chan bool)
    totalURLCount = 0

    urls = append(urls, "http://www.mastergoco.com/index1.html")
    urls = append(urls, "http://www.mastergoco.com/index2.html")
    urls = append(urls, "http://www.mastergoco.com/index3.html")
    urls = append(urls, "http://www.mastergoco.com/index4.html")
    urls = append(urls, "http://www.mastergoco.com/index5.html")

    fmt.Println("Starting spider")

    urlsProcessed = 0
    totalURLCount = len(urls)

    go evaluateStatus(statusChannel, textChannel, processChannel)

    go readURLs(statusChannel, textChannel)

    go addToScrapedText(textChannel, processChannel)

    for {
        if applicationStatus == false {
            fmt.Println(fullText)
            fmt.Println("Done!")
        }
    }
}
```

```

        break
    }
    select {
    case sC := <-statusChannel:
        fmt.Println("Message on StatusChannel", sC)

    }
}

```

This is a basic extrapolation of our last function, the capitalization function. However, each piece here is responsible for some aspect of reading URLs or appending its respective content to a larger variable.

In the following code, we created a sort of master loop that lets you know when a URL has been grabbed on `statusChannel`:

```

for {
    if applicationStatus == false {
        fmt.Println(fullText)
        fmt.Println("Done!")
        break
    }
    select {
    case sC := <- statusChannel:
        fmt.Println("Message on StatusChannel",sC)

    }
}

```

Often, you'll see this wrapped in `go func()` as part of a `WaitGroup` struct, or not wrapped at all (depending on the type of feedback you require).

The control flow, in this case, is `evaluateStatus`, which works as a channel monitor that lets us know when data crosses each channel and ends execution when it's complete. The `readURLs` function immediately begins reading our URLs, extracting the underlying data and passing it on to `textChannel`. At this point, our `addToScrapedText` function takes each sent HTML file and appends it to the `fullText` variable. When `evaluateStatus` determines that all URLs have been read, it sets `applicationStatus` to `false`. At this point, the infinite loop at the bottom of `main()` quits.

As mentioned, a crawler cannot come more rudimentary than this, but seeing a real-world example of how goroutines can work in congress will set us up for safer and more complex examples in the coming chapters.

Summary

In this chapter, we learned how to go from simple goroutines and instantiating channels to extending the basic functionality of goroutines and allowing cross-channel, bidirectional communication within concurrent processes. We looked at new ways to create blocking code to prevent our main process from ending before our goroutines. Finally, we learned about using select statements to develop reactive channels that are silent unless data is sent along a channel.

In our rudimentary web spider example, we employed these concepts together to create a safe, lightweight process that could extract all links from an array of URLs, grab the content via HTTP, and store the resulting response.

In the next chapter, we'll go beneath the surface to see how Go's internal scheduling manages concurrency and start using channels to really utilize the power, thrift, and speed of concurrency in Go.

2

Understanding the Concurrency Model

Now that we have a sense of what Go is capable of and how to test drive some concurrency models, we need to look deeper into Go's most powerful features to understand how to best utilize various concurrent tools and models.

We played with some general and basic goroutines to see how we can run concurrent processes, but we need to see how Go manages scheduling in concurrency before we get to communication between channels.

Understanding the working of goroutines

By this point, you should be well-versed in what goroutines do, but it's worth understanding *how* they work internally in Go. Go handles concurrency with cooperative scheduling, which, as we mentioned in the previous chapter, is heavily dependent on some form of blocking code.

The most common alternative to cooperative scheduling is preemptive scheduling, wherein each subprocess is granted a space of time to complete and then its execution is paused for the next.

Without some form of yielding back to the main thread, execution runs into issues. This is because Go works with a single process, working as a conductor for an orchestra of goroutines. Each subprocess is responsible to announce its own completion. As compared to other concurrency models, some of which allow for direct, named communication, this might pose a sticking point, particularly if you haven't worked with channels before.

You can probably see a potential for deadlocks given these facts. In this chapter, we'll discuss both the ways Go's design allows us to manage this and the methods to mitigate issues in applications wherein it fails.

Synchronous versus asynchronous goroutines

Understanding the concurrency model is sometimes an early pain point for programmers — not just for Go, but across languages that use different models as well. Part of this is due to operating in a *black box* (depending on your terminal preferences); a developer has to rely on logging or errors with data consistency to discern asynchronous and/or multiple core timing issues.

As the concepts of synchronous and asynchronous or concurrent and nonconcurrent tasks can sometimes be a bit abstract, we will have a bit of fun here in an effort to demonstrate all the concepts we've covered so far in a visual way.

There are, of course, a myriad of ways to address feedback and logging. You can write to files in `console/terminal/stdout...`, most of which are inherently linear in nature. There is no concise way to represent concurrency in a logfile. Given this and the fact that we deal with an emerging language with a focus on servers, let's take a different angle.

Instead of simply outputting to a file, we'll create a visual feedback that shows when a process starts and stops on a timeline.

Designing the web server plan

To show how approaches differ, we'll create a simple web server that loops through three trivial tasks and outputs their execution marks on an X-second timeline. We'll do this using a third-party library called `svggo` and the built-in `http` package for Go.

To start, let's grab the `svggo` library via `go get`:

```
go get github.com/ajstarks/svggo
```

If you try to install a package via the `go get` command and get an error about `$GOPATH` not being set, you need to set that environment variable. `GOPATH` is where Go will look to find installed import packages.

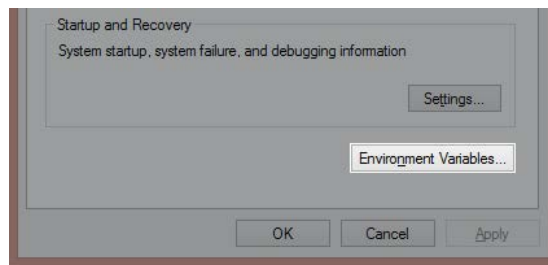
To set this in Linux (or Mac), type the following in bash (or Terminal):

```
export GOPATH=/usr/yourpathhere
```

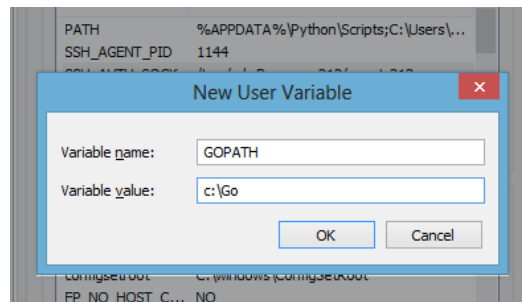
This path is up to you, so pick a place where you're most comfortable storing your Go packages.

To ensure it's globally accessible, install it where your Go binary is installed.

On Windows, you can right-click on **My Computer** and navigate to **Properties** | **Advanced system settings** | **Environment Variables...**, as shown in the following screenshot:



Here, you'll need to create a new variable called `GOPATH`. As with the Linux and Mac instructions, this can either be your Go language root directory or someplace else entirely. In this example, we've used `C:\Go`, as shown in the following screenshot:



Note that after taking these steps, you may need to reopen the Terminal, Command Prompt, or bash sessions before the value is read as valid. On *nix systems, you can log in and log out to initiate this.

Now that we have installed `gosvg`, we can visually demonstrate how the asynchronous and synchronous processes will look side-by-side as well as with multiple processors.

More libraries

Why SVG? We didn't need to use SVG and a web server, of course, and if you'd rather see an image generated and open that separately, there are other alternatives to do so. There are some additional graphical libraries available for Go, which are as follows:



- **draw2d:** As the name suggests, this is a two-dimensional drawing library for doing vector-style and raster graphics, which can be found at <https://code.google.com/p/draw2d/>.
- **graphics-go:** This project involves some members of the Go team itself. It's fairly limited in scope. You can find more about it at <https://code.google.com/p/graphics-go/>.
- **go:engine:** This is one of the few OpenGL implementations for Go. It can be overkill for this project, but if you find yourself in need of a three-dimensional graphics library, start at <http://go-engine.com/>.
- **Go-SDL:** Another possible overkill method, this is an implementation of the wonderful multimedia library SDL. You can find more about it at <https://github.com/banthar/Go-SDL>.

Robust GUI toolkits are also available, but as they were designed as systems languages, it isn't really Go's forte.

Visualizing concurrency

Our first attempt at visualizing concurrency will have two simple goroutines running the `drawPoint` function in a loop with 100 iterations. After running this, you can visit `localhost:1900/visualize` and see what concurrent goroutines look like.

If you run into problems with port 1900 (either with your firewall or through a port conflict), feel free to change the value on line 99 in the `main()` function. You may also need to access it through `127.0.0.1` if your system doesn't resolve `localhost`.

Note that we're not using `WaitGroup` or anything to manage the end of the goroutines because all we want to see is a visual representation of our code running. You can also handle this with a specific blocking code or `runtime.Gosched()`, as shown:

```
package main

import (
    "github.com/ajstarks/svgo"
    "net/http"
    "fmt"
    "log"
    "time"
    "strconv"
)

var width = 800
var height = 400
var startTime = time.Now().UnixNano()

func drawPoint(osvg *svg.SVG, pnt int, process int) {
    sec := time.Now().UnixNano()
    diff := (int64(sec) - int64(startTime)) / 100000

    pointLocation := 0

    pointLocation = int(diff)
    pointLocationV := 0
    color := "#000000"
    switch {
        case process == 1:
            pointLocationV = 60
            color = "#cc6666"
        default:
            pointLocationV = 180
            color = "#66cc66"
    }
}
```

```
    osvg.Rect(pointLocation,pointLocationV,3,5,"fill:"+color+";stroke:
none;")
    time.Sleep(150 * time.Millisecond)
}

func visualize(rw http.ResponseWriter, req *http.Request) {
    startTime = time.Now().UnixNano()
    fmt.Println("Request to /visualize")
    rw.Header().Set("Content-Type", "image/svg+xml")

    outputSVG := svg.New(rw)

    outputSVG.Start(width, height)
    outputSVG.Rect(10, 10, 780, 100, "fill:#eeeeee;stroke:none")
    outputSVG.Text(20, 30, "Process 1 Timeline", "text-
        anchor:start;font-size:12px;fill:#333333")
    outputSVG.Rect(10, 130, 780, 100, "fill:#eeeeee;stroke:none")
    outputSVG.Text(20, 150, "Process 2 Timeline", "text-
        anchor:start;font-size:12px;fill:#333333")

    for i:= 0; i < 801; i++ {
        timeText := strconv.FormatInt(int64(i),10)
        if i % 100 == 0 {
            outputSVG.Text(i,380,timeText,"text-anchor:middle;font-
                size:10px;fill:#000000")
        }else if i % 4 == 0 {
            outputSVG.Circle(i,377,1,"fill:#cccccc;stroke:none")
        }

        if i % 10 == 0 {
            outputSVG.Rect(i,0,1,400,"fill:#dddddd")
        }
        if i % 50 == 0 {
            outputSVG.Rect(i,0,1,400,"fill:#cccccc")
        }

    }

    for i := 0; i < 100; i++ {
        go drawPoint(outputSVG,i,1)
        drawPoint(outputSVG,i,2)
    }
}
```

```

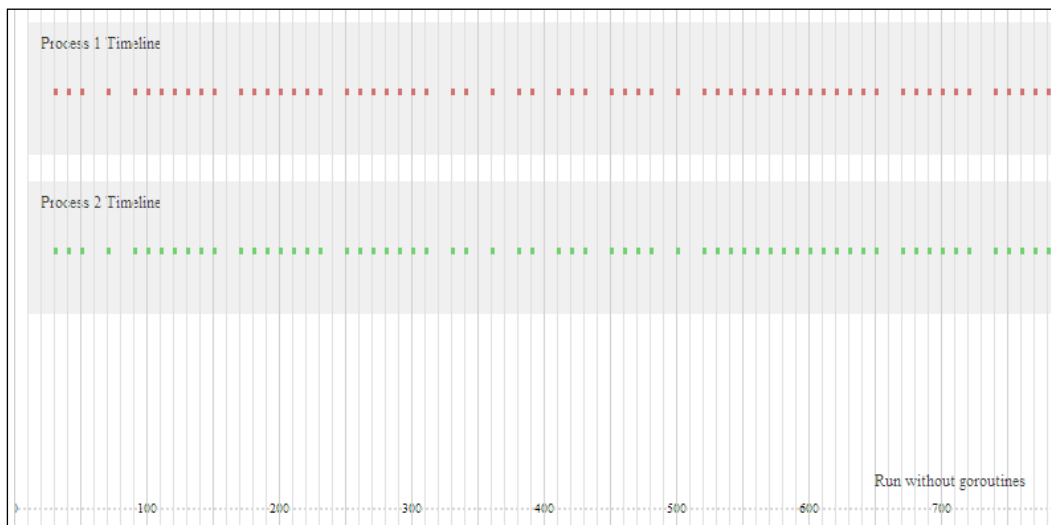
    outputSVG.Text(650, 360, "Run without goroutines", "text-
        anchor:start;font-size:12px;fill:#333333")
    outputSVG.End()
}

func main() {
    http.Handle("/visualize", http.HandlerFunc(visualize))

    err := http.ListenAndServe(":1900", nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

```

When you go to `localhost:1900/visualize`, you should see something like the following screenshot:



As you can see, everything is definitely running concurrently – our briefly sleeping goroutines hit on the timeline at the same moment. By simply forcing the goroutines to run in a serial fashion, you'll see a predictable change in this behavior. Remove the goroutine call on line 73, as shown:

```

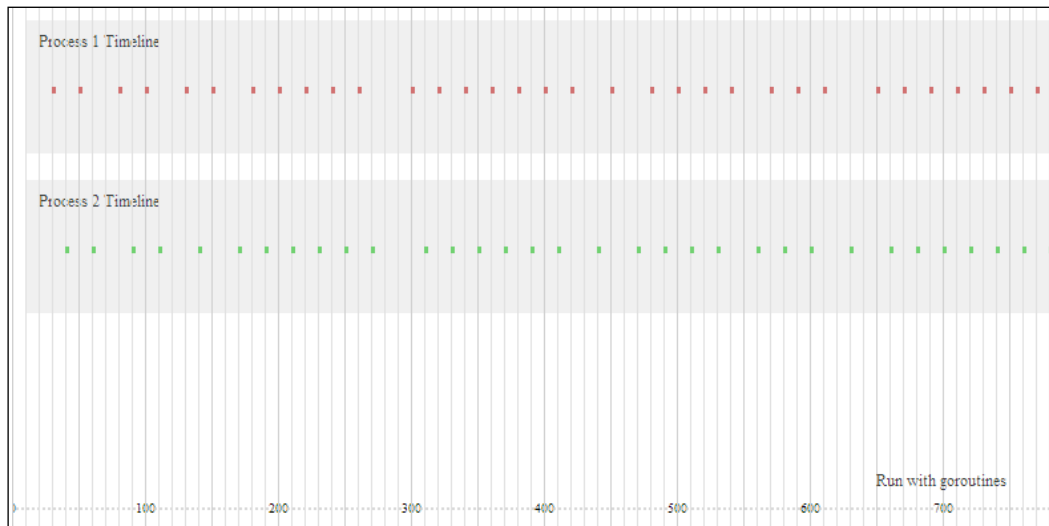
    drawPoint(outputSVG, i, 1)
    drawPoint(outputSVG, i, 2)

```

To keep our demonstration clean, change line 77 to indicate that there are no goroutines as follows:

```
outputSVG.Text(650, 360, "Run with goroutines", "text-  
anchor:start;font-size:12px;fill:#333333")
```

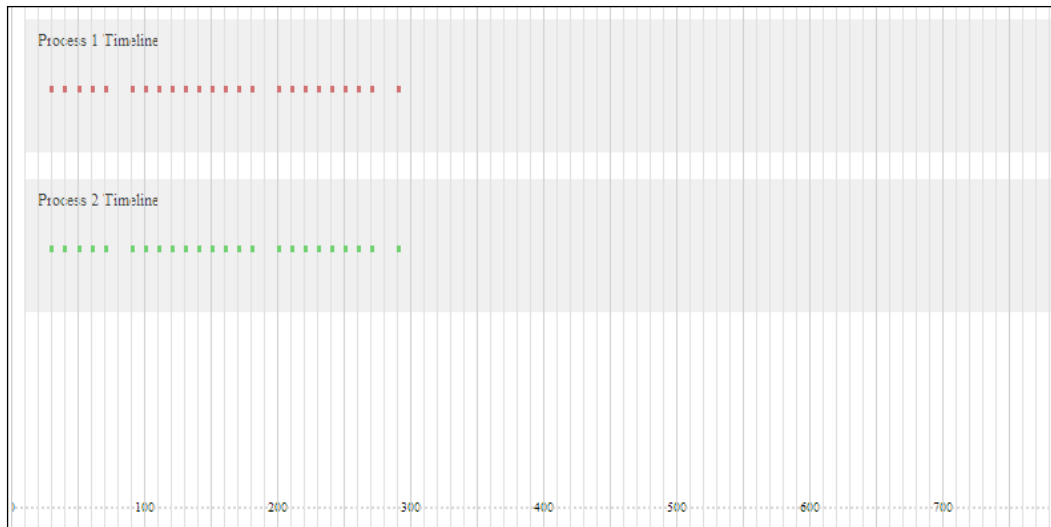
If we stop our server and restart with `go run`, we should see something like the following screenshot:



Now, each process waits for the previous process to complete before beginning. You can actually add this sort of feedback to any application if you run into problems with syncing data, channels, and processes.

If we so desired, we could add some channels and show communication across them as represented. Later, we will design a self-diagnosing server that gives real-time analytics about the state and status of the server, requests, and channels.

If we turn the goroutine back on and increase our maximum available processors, we'll see something similar to the following screenshot, which is not exactly the same as our first screenshot:



Your mileage will obviously vary depending on server speeds, the number of processors, and so on. But in this case, our change here resulted in a faster total execution time for our two processes with intermittent sleeps. This should come as no surprise, given we have essentially twice the bandwidth available to complete the two tasks.

RSS in action

Let's take the concept of **Rich Site Summary / Really Simple Syndication (RSS)** and inject some real potential delays to identify where we can best utilize goroutines in an effort to speed up execution and prevent blocking code. One common way to bring real-life, potentially blocking application elements into your code is to use something involving network transmission.

This is also a great place to look at timeouts and close channels to ensure that our program doesn't fall apart if something takes too long.

To accomplish both these requirements, we'll build a very basic RSS reader that will simply parse through and grab the contents of five RSS feeds. We'll read each of these as well as the provided links on each, and then we'll generate an SVG report of the process available via HTTP.



This is obviously an application best suited for a background task—you'll notice that each request can take a long time. However, for graphically representing a real-life process working with and without concurrency, it will work, especially with a single end user. We'll also log our steps to standard output, so be sure to take a look at your console as well.

For this example, we'll again use a third-party library, although it's entirely possible to parse RSS using Go's built-in XML package. Given the open-ended nature of XML and the specificity of RSS, we'll bypass them and use `go-pkg-rss` by Jim Teeuwen, available via the following `go get` command:

```
go get github.com/jteeuwen/go-pkg-rss
```

While this package is specifically intended as a replacement for the Google Reader product, which means that it does interval-based polling for new content within a set collection of sources, it also has a fairly neat and tidy RSS reading implementation. There are a few other RSS parsing libraries out there, though, so feel free to experiment.

An RSS reader with self diagnostics

Let's take a look at what we've learned so far, and use it to fetch and parse a set of RSS feeds concurrently while returning some visual feedback about the process in an internal web browser, as shown in the following code:

```
package main

import (
    "github.com/ajstarks/svgo"
    rss "github.com/jteeuwen/go-pkg-rss"
    "net/http"
    "log"
    "fmt"
    "strconv"
    "time"
    "os"
    "sync"
    "runtime"
)

type Feed struct {
    url string
    status int
    itemCount int
}
```

```
    complete bool
    itemsComplete bool
    index int
}
```

Here is the basis of our feed's overall structure: we have a `url` variable that represents the feed's location, a `status` variable to indicate whether it's started, and a `complete` Boolean variable to indicate it's finished. The next piece is an individual `FeedItem`; here's how it can be laid out:

```
type FeedItem struct {
    feedIndex int
    complete bool
    url string
}
```

Meanwhile, we will not do much with individual items; at this point, we simply maintain a URL, whether it's complete or a `FeedItem` struct's index.

```
var feeds []Feed
var height int
var width int
var colors []string
var startTime int64
var timeout int
var feedSpace int

var wg sync.WaitGroup

func grabFeed(feed *Feed, feedChan chan bool, osvg *svg.SVG) {

    startGrab := time.Now().Unix()
    startGrabSeconds := startGrab - startTime

    fmt.Println("Grabbing feed", feed.url, "
        at", startGrabSeconds, "second mark")

    if feed.status == 0 {
        fmt.Println("Feed not yet read")
        feed.status = 1

        startX := int(startGrabSeconds * 33);
        startY := feedSpace * (feed.index)

        fmt.Println(startY)
        wg.Add(1)
```

```

rssFeed := rss.New(timeout, true, channelHandler,
    itemsHandler);

if err := rssFeed.Fetch(feed.url, nil); err != nil {
    fmt.Fprintf(os.Stderr, "[e] %s: %s", feed.url, err)
    return
} else {

    endSec := time.Now().Unix()
    endX := int( (endSec - startGrab) )
    if endX == 0 {
        endX = 1
    }
    fmt.Println("Read feed in",endX,"seconds")
    osvg.Rect(startX,startY,endX,feedSpace,"fill:
        #000000;opacity:.4")
    wg.Wait()

    endGrab := time.Now().Unix()
    endGrabSeconds := endGrab - startTime
    feedEndX := int(endGrabSeconds * 33);

    osvg.Rect(feedEndX,startY,1,feedSpace,"fill:#ff0000;opacity:.9")

    feedChan <- true
}

}else if feed.status == 1{
    fmt.Println("Feed already in progress")
}

}

```

The `grabFeed()` method directly controls the flow of grabbing any individual feed. It also bypasses potential concurrent duplication through the `WaitGroup` struct. Next, let's check out the `itemsHandler` function:

```
func channelHandler(feed *rss.Feed, newchannels []*rss.Channel) {
}

func itemsHandler(feed *rss.Feed, ch *rss.Channel, newitems
    []*rss.Item) {
```

```

fmt.Println("Found", len(newitems), "items in", feed.Url)

for i := range newitems {
    url := *newitems[i].Guid
    fmt.Println(url)
}

wg.Done()
}

```

The `itemsHandler` function doesn't do much at this point, other than instantiating a new `FeedItem` struct—in the real world, we'd take this as the next step and retrieve the values of the items themselves. Our next step is to look at the process that grabs individual feeds and marks the time taken for each one, as follows:

```

func getRSS(rw http.ResponseWriter, req *http.Request) {
    startTime = time.Now().Unix()
    rw.Header().Set("Content-Type", "image/svg+xml")
    outputSVG := svg.New(rw)
    outputSVG.Start(width, height)

    feedSpace = (height-20) / len(feeds)

    for i:= 0; i < 30000; i++ {
        timeText := strconv.FormatInt(int64(i/10),10)
        if i % 1000 == 0 {
            outputSVG.Text(i/30,390,timeText,"text-anchor:middle;font-size:10px;fill:#000000")
        } else if i % 4 == 0 {
            outputSVG.Circle(i,377,1,"fill:#cccccc;stroke:none")
        }

        if i % 10 == 0 {
            outputSVG.Rect(i,0,1,400,"fill:#dddddd")
        }
        if i % 50 == 0 {
            outputSVG.Rect(i,0,1,400,"fill:#cccccc")
        }
    }
}

```

```
feedChan := make(chan bool, 3)

for i := range feeds {

    outputSVG.Rect(0, (i*feedSpace), width, feedSpace,
        "fill:"+colors[i]+";stroke:none;")
    feeds[i].status = 0
    go grabFeed(&feeds[i], feedChan, outputSVG)
    <- feedChan
}

outputSVG.End()
}
```

Here, we retrieve the RSS feed and mark points on our SVG with the status of our retrieval and read events. Our `main()` function will primarily handle the setup of feeds, as follows:

```
func main() {

    runtime.GOMAXPROCS(2)

    timeout = 1000

    width = 1000
    height = 400

    feeds = append(feeds, Feed{index: 0, url:
        "https://groups.google.com/forum/feed/golang-
        nuts/messages/rss_v2_0.xml?num=50", status: 0, itemCount: 0,
        complete: false, itemsComplete: false})
    feeds = append(feeds, Feed{index: 1, url:
        "http://www.reddit.com/r/golang/.rss", status: 0, itemCount:
        0, complete: false, itemsComplete: false})
    feeds = append(feeds, Feed{index: 2, url:
        "https://groups.google.com/forum/feed/golang-
        dev/messages/rss_v2_0.xml?num=50", status: 0, itemCount: 0,
        complete: false, itemsComplete: false })
}
```

Here is our slice of `FeedItem` structs:

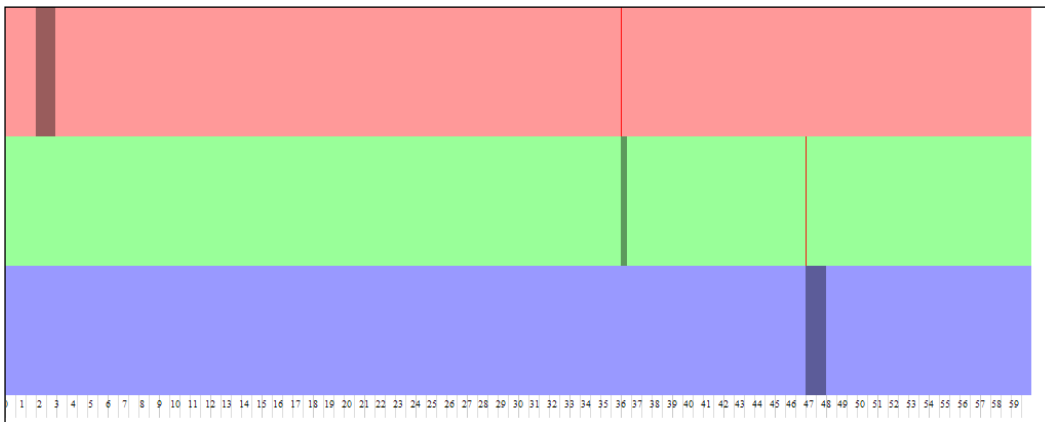
```
colors = append(colors, "#ff9999")
colors = append(colors, "#99ff99")
colors = append(colors, "#9999ff")
```

In the print version, these colors may not be particularly useful, but testing it on your system will allow you to delineate between events inside the application. We'll need an HTTP route to act as an endpoint; here's how we'll set that up:

```
http.Handle("/getrss", http.HandlerFunc(getRSS))
err := http.ListenAndServe(":1900", nil)
if err != nil {
    log.Fatal("ListenAndServe:", err)
}
```

When run, you should see the start and duration of the RSS feed retrieval and parsing, followed by a thin line indicating that the feed has been parsed and all items read.

Each of the three blocks expresses the full time to process each feed, demonstrating the nonconcurrent execution of this version, as shown in the following screenshot:



Note that we don't do anything interesting with the feed items, we simply read the URL. The next step will be to grab the items via HTTP, as shown in the following code snippet:

```
url := *newitems[i].Guid
response, _, err := http.Get(url)
if err != nil {
    // ...
}
```

With this example, we stop at every step to provide some sort of feedback to the SVG that some event has occurred. Our channel here is buffered and we explicitly state that it must receive three Boolean messages before it can finish blocking, as shown in the following code snippet:

```
feedChan := make(chan bool, 3)

for i := range feeds {

    outputSVG.Rect(0, (i*feedSpace), width, feedSpace,
        "fill:"+colors[i]+";stroke:none;")
    feeds[i].status = 0
    go grabFeed(&feeds[i], feedChan, outputSVG)
    <- feedChan
}

outputSVG.End()
```

By giving 3 as the second parameter in our channel invocation, we tell Go that this channel must receive three responses before continuing the application. You should use caution with this, though, particularly in setting things explicitly as we have done here. What if one of the goroutines never sent a Boolean across the channel? The application would crash.

Note that we also increased our timeline here, from 800ms to 60 seconds, to allow for retrieval of all feeds. Keep in mind that if our script exceeds 60 seconds, all actions beyond that time will occur outside of this visual timeline representation.

By implementing the `WaitGroup` struct while reading feeds, we impose some serialization and synchronization to the application. The second feed will not start until the first feed has completed retrieving all URLs. You can probably see where this might introduce some errors going forward:

```
wg.Add(1)
rssFeed := rss.New(timeout, true, channelHandler,
    itemsHandler);
...
wg.Wait()
```

This tells our application to yield until we set the `Done()` command from the `itemsHandler()` function.

So what happens if we remove `WaitGroups` entirely? Given that the calls to grab the feed items are asynchronous, we may not see the status of all of our RSS calls; instead, we might see just one or two feeds or no feed at all.

Imposing a timeout

So what happens if nothing runs within our timeline? As you might expect, we'll get three bars with no activity in them. It's important to consider how to kill processes that aren't doing what we expect them to. In this case, the best method is a timeout. The `Get` method in the `http` package does not natively support a timeout, so you'll have to roll your own `rssFeed.Fetch` (and underlying `http.Get()`) implementation if you want to prevent these requests from going into perpetuity and killing your application. We'll dig into this a bit later; in the mean time, take a look at the `Transport` struct, available in the core `http` package at <http://golang.org/pkg/net/http/#Transport>.

A little bit about CSP

We touched on CSP briefly in the previous chapter, but it's worth exploring a bit more in the context of how Go's concurrency model operates.

CSP evolved in the late 1970s and early 1980s through the work of Sir Tony Hoare and is still in the midst of evolution today. Go's implementation is heavily based on CSP, but it neither entirely follows all the rules and conventions set forth in its initial description nor does it follow its evolution since.

One of the ways in which Go differs from true CSP is that as it is defined, a process in Go will only continue so long as there exists a channel ready to receive from that process. We've already encountered a couple of deadlocks that were the result of a listening channel with nothing to receive. The inverse is also true; a deadlock can result from a channel continuing without sending anything, leaving its receiving channel hanging indefinitely.

This behavior is endemic to Go's scheduler, and it should really only pose problems when you're working with channels initially.



Hoare's original work is now available (mostly) free from a number of institutions. You can read, cite, copy, and redistribute it free of charge (but not for commercial gain). If you want to read the whole thing, you can grab it at <http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf>.

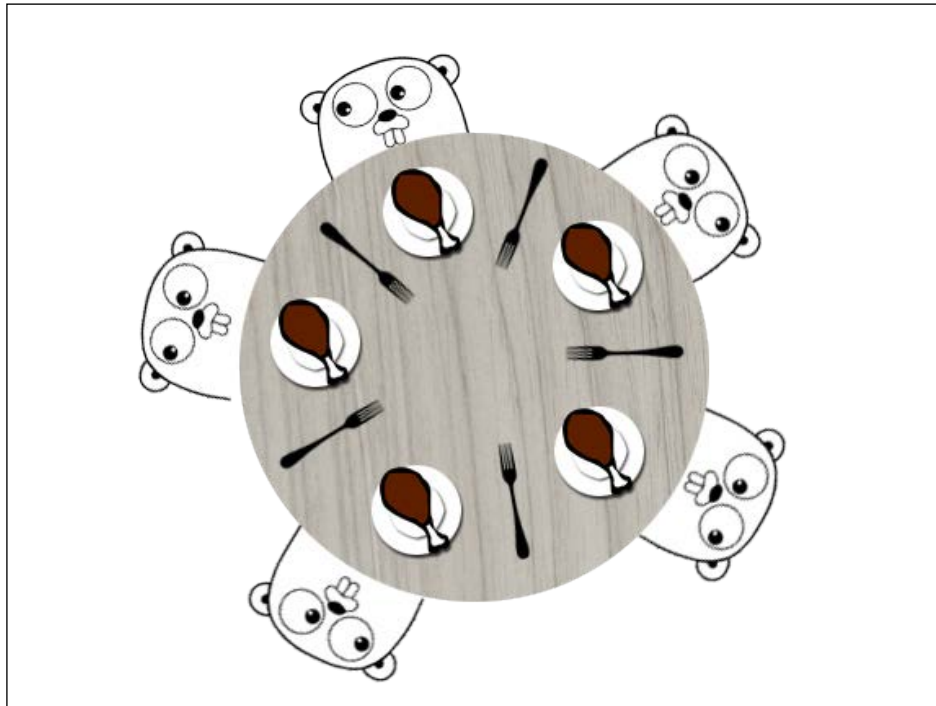
The complete book itself is also available at <http://www.usingcsp.com/cspbook.pdf>.

As of this publishing, Hoare is working as a researcher at Microsoft.

As per the designers of the application itself, the goal of Go's implementation of CSP concepts was to focus on simplicity – you don't have to worry about threads or mutexes unless you really want to or need to.

The dining philosophers problem

You may have heard of the dining philosophers problem, which describes the kind of problems concurrent programming was designed to solve. The dining philosophers problem was formulated by the great Edsger Dijkstra. The crux of the problem is a matter of resources – five philosophers sit at a table with five plates of food and five forks, and each can only eat when he has two forks (one to his left and another to his right). A visual representation is shown as follows:



With a single fork on either side, any given philosopher can only eat when he has a fork in both hands and must put both back on the table when complete. The idea is to coordinate the meal such that all of the philosophers can eat in perpetuity without any starving – two philosophers must be able to eat at any moment and there can be no deadlocks. They're philosophers because when they're not eating, they're thinking. In a programming analog, you can consider this as either a waiting channel or a sleeping process.

Go handles this problem pretty succinctly with goroutines. Given five philosophers (in an individual struct, for example), you can have all five alternate between thinking, receiving a notification when the forks are down, grabbing forks, dining with forks, and placing the forks down.

Receiving the notification that the forks are down acts as the listening channel, dining and thinking are separate processes, and placing the forks down operates as an announcement along the channel.

We can visualize this concept in the following pseudo Go code:

```
type Philosopher struct {
    leftHand bool
    rightHand bool
    status int
    name string
}

func main() {

    philosophers := [...]Philosopher{"Kant", "Turing",
        "Descartes", "Kierkegaard", "Wittgenstein"}

    evaluate := func() {
        for {

            select {
                case <- forkUp:
                    // philosophers think!
                case <- forkDown:
                    // next philosopher eats in round robin
            }

        }
    }

}
```

This example has been left very abstract and nonoperational so that you have a chance to attempt to solve it. We will build a functional solution for this in the next chapter, so make sure to compare your solution later on.

There are hundreds of ways to handle this problem, and we'll look at a couple of alternatives and how they can or cannot play nicely within Go itself.

Go and the actor model

The actor model is something that you'll likely be very familiar with if you're an Erlang or Scala user. The difference between CSP and the actor model is negligible but important. With CSP, messages from one channel can only be completely sent if another channel is listening and ready for them. The actor model does not necessarily require a ready channel for another to send. In fact, it stresses direct communication rather than relying on the conduit of a channel.

Both systems can be nondeterministic, which we've already seen demonstrated in Go/CSP in our earlier examples. CSP and goroutines are anonymous and transmission is specified by the channel rather than the source and destination. An easy way to visualize this in pseudocode in the actor model is as follows:

```
a = new Actor
b = new Actor
a -> b("message")
```

In CSP, it is as follows:

```
a = new Actor
b = new Actor
c = new Channel
a -> c("sending something")
b <- c("receiving something")
```

Both serve the same fundamental functionality but through slightly different ways.

Object orientation

As you work with Go, you will notice that there is a core characteristic that's often espoused, which users may feel is wrong. You'll hear that Go is not an object-oriented language, and yet you have structs that can have methods, those methods in turn can have methods, and you can have communication to and from any instantiation of it. Channels themselves may feel like primitive object interfaces, capable of setting and receiving values from a given data element.

The message passing implementation of Go is, indeed, a core concept of object-oriented programming. Structs with interfaces operate essentially as classes, and Go supports polymorphism (although not parametric polymorphism). Yet, many who work with the language (and who have designed it) stress that it is not object oriented. So what gives?

Much of this definition ultimately depends on who you ask. Some believe that Go lacks some of the requisite characteristics of object-oriented programming, and others believe it satisfies them. The most important thing to keep in mind is that you're not limited by Go's design. Anything that you can do in a *true* object-oriented language can be handled without much struggle within Go.

Demonstrating simple polymorphism in Go

As mentioned before, if you expect polymorphism to resemble object-oriented programming, this may not represent a syntactical analogue. However, the use of interfaces as an abstraction of class-bound polymorphic methods is just as clean, and in many ways, more explicit and readable. Let's look at a very simple implementation of polymorphism in Go:

```
type intInterface struct {  
  
}  
  
type stringInterface struct {  
  
}  
  
func (number intInterface) Add (a int, b int) int {  
    return a + b;  
}  
  
func (text stringInterface) Add (a string, b string) string {  
    return a + b  
}  
  
func main() {  
  
    number := new (intInterface)  
    fmt.Println( number.Add(1,2) )  
  
    text := new (stringInterface)  
    fmt.Println( text.Add("this old man", " he played one"))  
  
}
```

As you can see, we use an interface (or its Go analog) to disambiguate methods. You cannot have generics the same way you might in Java, for example. This, however, boils down to a mere matter of style in the end. You should neither find this daunting nor will it impose any cruft or ambiguity into your code.

Using concurrency

It hasn't yet been mentioned, but we should be aware that concurrency is not always necessary and beneficial for an application. There exists no real rule of thumb, and it's rare that concurrency will introduce problems to an application; but if you really think about applications as a whole, not all will require concurrent processes.

So what works best? As we've seen in the previous example, anything that introduces potential latency or I/O blocking, such as network calls, disk reads, third-party applications (primarily databases), and distributed systems, can benefit from concurrency. If you have the ability to do work while other work is being done on an undetermined timeline, concurrency strategies can improve the speed and reliability of an application.

The lesson here is you should never feel compelled to shoehorn concurrency into an application that doesn't really require it. Programs with inter-process dependencies (or lack of blocking and external dependencies) may see little or no benefit from implementing concurrency structures.

Managing threads

So far, you've probably noticed that thread management is not a matter that requires the programmer's utmost concern in Go. This is by design. Goroutines aren't tied to a specific thread or threads that are handled by Go's internal scheduler. However, this doesn't mean that you neither have access to the threads nor the ability to control what individual threads do. As you know, you can already tell Go how many threads you have (or wish to use) by using `GOMAXPROCS`. We also know that using this can introduce asynchronous issues as it pertains to data consistency and execution order.

At this point, the main issue with threads is not how they're accessed or utilized, but how to properly control execution flow to guarantee that your data is predictable and synchronized.

Using sync and mutexes to lock data

One issue that you may have run into with the preceding examples is the notion of atomic data. After all, if you deal with variables and structures across multiple goroutines, and possibly processors, how do you ensure that your data is safe across them? If these processes run in parallel, coordinating data access can sometimes be problematic.

Go provides a bevy of tools in its `sync` package to handle these types of problems. How elegantly you approach them depends heavily on your approach, but you should never have to reinvent the wheel in this realm.

We've already looked at the `WaitGroup` struct, which provides a simple method to tell the main thread to pause until the next notification that says a waiting process has done what it's supposed to do.

Go also provides a direct abstraction to a mutex. It may seem contradictory to call something a direct abstraction, but the truth is you don't have access to Go's scheduler, only an approximation of a true mutex.

We can use a mutex to lock and unlock data and guarantee atomicity in our data. In many cases, this may not be necessary; there are a great many times where the order of execution does not impact the consistency of the underlying data. However, when we do have concerns about this value, it's helpful to be able to invoke a lock explicitly. Let's take the following example:

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    current := 0
    iterations := 100
    wg := new (sync.WaitGroup);

    for i := 0; i < iterations; i++ {
        wg.Add(1)
```

```
        go func() {
            current++
            fmt.Println(current)
            wg.Done()
        }()
        wg.Wait()
    }
}
```

Unsurprisingly, this provides a list of 0 to 99 in your terminal. What happens if we change `WaitGroup` to know there will be 100 instances of `Done()` called, and put our blocking code at the end of the loop?

To demonstrate a simple proposition of why and how to best utilize `waitGroups` as a mechanism for concurrency control, let's do a simple number iterator and look at the results. We will also check out how a directly called mutex can augment this functionality, as follows:

```
func main() {
    runtime.GOMAXPROCS(2)
    current := 0
    iterations := 100
    wg := new(sync.WaitGroup);
    wg.Add(iterations)
    for i := 0; i < iterations; i++ {
        go func() {
            current++
            fmt.Println(current)
            wg.Done()
        }()
    }
    wg.Wait()
}
```

Now, our order of execution is suddenly off. You may see something like the following output:

```
95
96
98
```

```
99
100
3
4
```

We have the ability to lock and unlock the current command at will; however, this won't change the underlying execution order, it will only prevent reading and/or writing to and from a variable until an unlock is called.

Let's try to lock down the variable we're outputting using `mutex`, as follows:

```
for i := 0; i < iterations; i++ {
    go func() {
        mutex.Lock()
        fmt.Println(current)
        current++
        mutex.Unlock()
        fmt.Println(current)
        wg.Done()
    }()
}
```

You can probably see how a mutex control mechanism can be important to enforce data integrity in your concurrent application. We'll look more at mutexes and locking and unlocking processes in *Chapter 4, Data Integrity in an Application*.

Summary

In this chapter, we've tried to remove some of the ambiguity of Go's concurrency patterns and models by giving visual, real-time feedback to a few applications, including a rudimentary RSS aggregator and reader. We examined the dining philosophers problem and looked at ways you can use the Go concurrency topics to solve the problem neatly and succinctly. We compared the way CSP and actor models are similar and ways in which they differ.

In the next chapter, we will take these concepts and apply them to the process of developing a strategy to maintain concurrency in an application.

3

Developing a Concurrent Strategy

In the previous chapter, we looked at the concurrency model that Go relies on to make your life as a developer easier. We also saw a visual representation of parallelism and concurrency. These help us to understand the differences and overlaps between serialized, concurrent, and parallel applications.

However, the most critical part of any concurrent application is not the concurrency itself but communication and coordination between the concurrent processes.

In this chapter, we'll look at creating a plan for an application that heavily factors communication between processes and how a lack of coordination can lead to significant issues with consistency. We'll look at ways we can visualize our concurrent strategy on paper so that we're better equipped to anticipate potential problems.

Applying efficiency in complex concurrency

When designing applications, we often eschew complex patterns for simplicity, with the assumption that simple systems are often the fastest and most efficient. It seems only logical that a machine with fewer moving parts will be more efficient than one with more.

The paradox here, as it applies to concurrency, is that adding redundancy and significantly more movable parts often leads to a more efficient application. If we consider concurrent schemes, such as goroutines, to be infinitely scalable resources, employing more should always result in some form of efficiency benefit. This applies not just to parallel concurrency but to single core concurrency as well.

If you find yourself designing an application that utilizes concurrency at the cost of efficiency, speed, and consistency, you should ask yourself whether the application truly needs concurrency at all.

When we talk about efficiency, we aren't just dealing with speed. Efficiency should also weigh the CPU and memory overhead and the cost to ensure data consistency.

For example, should an application marginally benefit from concurrency but require an elaborate and/or computationally expensive process to guarantee data consistency, it's worth re-evaluating the strategy entirely.

Keeping your data reliable and up to date should be paramount; while having unreliable data may not always have a devastating effect, it will certainly compromise the reliability of your application.

Identifying race conditions with race detection

If you've ever written an application that depends on the exact timing and sequencing of functions or methods to create a desired output, you're already quite familiar with race conditions.

These are particularly common anytime you deal with concurrency and far more so when parallelism is introduced. We've actually encountered a few of them in the first few chapters, specifically with our incrementing number function.

The most commonly used educational example of race conditions is that of a bank account. Assume that you start with \$1,000 and attempt 200 \$5 transactions. Each transaction requires a query on the current balance of the account. If it passes, the transaction is approved and \$5 is removed from the balance. If it fails, the transaction is declined and the balance remains unchanged.

This is all well and good until the query happens at some point during a concurrent transaction (in most cases in another thread). If, for example, a thread asks "Do you have \$5 in your account?" as another thread is in the process of removing \$5 but has not yet completed, you can end up with an approved transaction that should have been declined.

Tracking down the cause of race conditions can be — to say the least — a gigantic headache. With Version 1.1 of Go, Google introduced a race detection tool that can help you locate potential issues.

Let's take a very basic example of a multithreaded application with race conditions and see how Golang can help us debug it. In this example, we'll build a bank account that starts with \$1,000 and runs 100 transactions for a random amount between \$0 and \$25.

Each transaction will be run in its own goroutine, as follows:

```
package main

import (
    "fmt"
    "time"
    "sync"
    "runtime"
    "math/rand"
)

var balance int
var transactionNo int

func main() {
    rand.Seed(time.Now().Unix())
    runtime.GOMAXPROCS(2)
    var wg sync.WaitGroup

    tranChan := make(chan bool)

    balance = 1000
    transactionNo = 0
    fmt.Println("Starting balance: $",balance)

    wg.Add(1)
    for i := 0; i < 100; i++ {
        go func(ii int, trChan chan(bool)) {
            transactionAmount := rand.Intn(25)
            transaction(transactionAmount)
            if (ii == 99) {
                trChan <- true
            }
        }(i, tranChan)
    }
}
```

```
go transaction(0)
select {

    case <- tranChan:
        fmt.Println("Transactions finished")
        wg.Done()

}

wg.Wait()
close(tranChan)
fmt.Println("Final balance: $",balance)
}

func transaction(amt int) (bool) {

    approved := false
    if (balance-amt) < 0 {
        approved = false
    }else {
        approved = true
        balance = balance - amt
    }

    approvedText := "declined"
    if (approved == true) {
        approvedText = "approved"
    }else {

    }

    transactionNo = transactionNo + 1
    fmt.Println(transactionNo, "Transaction for $",amt,approvedText)
    fmt.Println("\tRemaining balance $",balance)
    return approved
}
```

Depending on your environment (and whether you enable multiple processors), you might have the previous goroutine operate successfully with a \$0 or more final balance. You might, on the other hand, simply end up with transactions that exceed the balance at the time of transaction, resulting in a negative balance.

So how do we know for sure?

For most applications and languages, this process often involves a lot of running, rerunning, and logging. It's not unusual for race conditions to present a daunting and laborious debugging process. Google knows this and has given us a race condition detection tool. To test this, simply use the `-race` flag when testing, building, or running your application, as shown:

```
go run -race race-test.go
```

When run on the previous code, Go will execute the application and then report any possible race conditions, as follows:

```
>> Final balance: $0
>> Found 2 data race(s)
```

Here, Go is telling us there are two potential race conditions with data. It isn't telling us that these will surely create data consistency issues, but if you run into such problems, this may give you some clue as to why.

If you look at the top of the output, you'll get more detailed notes on what's causing a race condition. In this example, the details are as follows:

```
=====
WARNING: DATA RACE
Write by goroutine 5: main.transaction()    /var/go/race.go:75 +0xbd
    main.func_001()    /var/go/race.go:31 +0x44

Previous write by goroutine 4: main.transaction()
    /var/go/race.go:75 +0xbd main.func_001()    /var/go/race.go:31
    +0x44

Goroutine 5 (running) created at: main.main()    /var/go/race.go:36
    +0x21c

Goroutine 4 (finished) created at: main.main()    /var/go/race.go:36
    +0x21c
```

We get a detailed, full trace of where our potential race conditions exist. Pretty helpful, huh?

The race detector is guaranteed to not produce false positives, so you can take the results as strong evidence that there is a potential problem in your code. The potential is stressed here because a race condition can go undetected in normal conditions very often—an application may work as expected for days, months, or even years before a race condition can surface.



We've mentioned logging, and if you aren't intimately familiar with Go's core language, your mind might go in a number of directions—stdout, file logs, and so on. So far we've stuck to stdout, but you can use the standard library to handle this logging. Go's log package allows you to write to io or stdout as shown:

```
messageOutput := os.Stdout
logOut := log.New(messageOutput, "Message: ", log.
Ldate|log.Ltime|log.Llongfile);
logOut.Println("This is a message from the
application!")
```

This will produce the following output:

```
Message: 2014/01/21 20:59:11 /var/go/log.go:12: This is
a message from the application!
```

So, what's the advantage of the log package versus rolling your own? In addition to being standardized, this package is also synchronized in terms of output.

So what now? Well, there are a few options. You can utilize your channels to ensure data integrity with a buffered channel, or you can use the `sync.Mutex` struct to lock your data.

Using mutual exclusions

Typically, mutual exclusion is considered a low-level and best-known approach to synchronicity in your application—you should be able to address data consistency within communication between your channels. However, there will be instances where you need to truly block read/write on a value while you work with it.

At the CPU level, a mutex represents an exchange of binary integer values across registers to acquire and release locks. We'll deal with something on a much higher level, of course.

We're already familiar with the `sync` package from our use of the `WaitGroup` struct, but the package also contains the conditional variables struct `Cond` and `Once`, which will perform an action just one time, and the mutual exclusion locks `RWMutex` and `Mutex`. As the name `RWMutex` implies, it is open to multiple readers and/or writers to lock and unlock; there is more on this later in this chapter and in *Chapter 5, Locks, Blocks, and Better Channels*.

All of these – as the package name implies – empower you to prevent race conditions on data that may be accessed by any number of goroutines and/or threads. Using any of the methods in this package does not ensure atomicity within data and structures, but it does give you the tools to manage atomicity effectively. Let's look at a few ways we can solidify our account balance in concurrent, threadsafe applications.

As mentioned previously, we can coordinate data changes at the channel level whether that channel is buffered or unbuffered. Let's offload the logic and data manipulation to the channel and see what the `-race` flag presents.

If we modify our main loop, as shown in the following code, to utilize messages received by the channel to manage the balance value, we will avoid race conditions:

```
package main

import (
    "fmt"
    "time"
    "sync"
    "runtime"
    "math/rand"
)

var balance int
var transactionNo int

func main() {
    rand.Seed(time.Now().Unix())
    runtime.GOMAXPROCS(2)
    var wg sync.WaitGroup
    balanceChan := make(chan int)
    tranChan := make(chan bool)

    balance = 1000
    transactionNo = 0
    fmt.Println("Starting balance: ", balance)

    wg.Add(1)
    for i := 0; i < 100; i++ {

        go func(ii int) {

            transactionAmount := rand.Intn(25)
            balanceChan <- transactionAmount
```



```
        if ii == 99 {
            fmt.Println("Should be quittin time")
            tranChan <- true
            close(balanceChan)
            wg.Done()
        }
    }(i)
}

go transaction(0)

breakPoint := false
for {
    if breakPoint == true {
        break
    }
    select {
        case amt:= <- balanceChan:
            fmt.Println("Transaction for $",amt)
            if (balance - amt) < 0 {
                fmt.Println("Transaction failed!")
            }else {
                balance = balance - amt
                fmt.Println("Transaction succeeded")
            }
            fmt.Println("Balance now $",balance)

        case status := <- tranChan:
            if status == true {
                fmt.Println("Done")
                breakPoint = true
                close(tranChan)
            }
    }
}

wg.Wait()

fmt.Println("Final balance: $",balance)
}
```

```

func transaction(amt int) (bool) {

    approved := false
    if (balance-amt) < 0 {
        approved = false
    }else {
        approved = true
        balance = balance - amt
    }

    approvedText := "declined"
    if (approved == true) {
        approvedText = "approved"
    }else {

    }
    transactionNo = transactionNo + 1
    fmt.Println(transactionNo,"Transaction for $",amt,approvedText)
    fmt.Println("\tRemaining balance $",balance)
    return approved
}

```

This time, we let the channel manage the data entirely. Let's look at what we're doing:

```

transactionAmount := rand.Intn(25)
balanceChan <- transactionAmount

```

This still generates a random integer between 0 and 25, but instead of passing it to a function, we pass the data along the channel. Channels allow you to control the ownership of data neatly. We then see the select/listener, which largely mirrors our `transaction()` function defined earlier in this chapter:

```

case amt:= <- balanceChan:
    fmt.Println("Transaction for $",amt)
    if (balance - amt) < 0 {
        fmt.Println("Transaction failed!")
    }else {
        balance = balance - amt
        fmt.Println("Transaction succeeded")
    }
    fmt.Println("Balance now $",balance)

```

To test whether we've averted a race condition, we can run `go run` with the `-race` flag again and see no warnings.

Channels can be seen as the sanctioned go-to way of handling synchronized dataUse `Sync.Mutex()`.

As mentioned, having a built-in race detector is a luxury not afforded to developers in most languages, and having it allows us to test methodologies and get real-time feedback on each.

We noted that using an explicit mutex is discouraged in favor of channels of goroutines. This isn't always exactly true because there is a right time and place for everything, and mutexes are no exclusion. What's worth noting is that mutexes are implemented internally by Go for channels. As was previously mentioned, you can use explicit channels to handle reads and writes and juggle the data between them.

However, this doesn't mean there is no use for explicit locks. An application that has many reads and very few writes might benefit from explicit locks for writes; this doesn't necessarily mean that the reads will be dirty reads, but it could result in faster and/or more concurrent execution.

For the sake of demonstration, let's remove our race condition using an explicit lock. Our `-race` flag tells us where it encounters read/write race conditions, as shown:

Read by goroutine 5: main.transaction() /var/go/race.go:62 +0x46

The previous line is just one among several others we'll get from the race detection report. If we look at line 62 in our code, we'll find a reference to `balance`. We'll also find a reference to `transactionNo`, our second race condition. The easiest way to address both is to place a mutual exclusion lock around the contents of the `transaction` function as this is the function that modifies the `balance` and `transactionNo` variables. The `transaction` function is as follows:

```
func transaction(amt int) (bool) {
    mutex.Lock()

    approved := false
    if (balance-amt) < 0 {
        approved = false
    }else {
        approved = true
        balance = balance - amt
    }

    approvedText := "declined"
    if (approved == true) {
        approvedText = "approved"
    }else {
```

```
}
transactionNo = transactionNo + 1
fmt.Println(transactionNo, "Transaction for $", amt, approvedText)
fmt.Println("\tRemaining balance $", balance)

mutex.Unlock()
return approved
}
```

We also need to define `mutex` as a global variable at the top of our application, as shown:

```
var mutex sync.Mutex
```

If we run our application now with the `-race` flag, we get no warnings.

The `mutex` variable is, for practical purposes, an alternative to the `WaitGroup` struct, which functions as a conditional synchronization mechanism. This is also the way that the channels operate—data that moves along channels is contained and isolated between goroutines. A channel can effectively work as a first-in, first-out tool in this way by binding goroutine state to `WaitGroup`; data accessed across the channel can then be provided safety via the lower-level `mutex`.

Another worthwhile thing to note is the versatility of a channel—we have the ability to share a channel among an array of goroutines to receive and/or send data, and as a first-class citizen, we can pass them along in functions.

Exploring timeouts

Another noteworthy thing we can do with channels is explicitly kill them after a specified amount of time. This is an operation that will be a bit more involved should you decide to manually handle mutual exclusions.

The ability to kill a long-running routine through the channel is extremely helpful; consider a network-dependent operation that should not only be restricted to a short time period but also not allowed to run for a long period. In other words, you want to offer the process a few seconds to complete; but if it runs for more than a minute, our application should know that something has gone wrong enough to stop attempting to listen or send on that channel. The following code demonstrates using a timeout channel in a `select` call:

```
func main() {

    ourCh := make(chan string, 1)
```

```
go func() {  
  
}()  
  
select {  
    case <-time.After(10 * time.Second):  
        fmt.Println("Enough's enough")  
        close(ourCh)  
}  
  
}
```

If we run the previous simple application, we'll see that our goroutine will be allowed to do nothing for exactly 10 seconds, after which we implement a timeout safeguard that bails us out.

You can see this as being particularly useful in network applications; even in the days of blocking and thread-dependent servers, timeouts like these were implemented to prevent a single misbehaving request or process to gum up the entire server. This is the very basis of a classic web server problem that we'll revisit in more detail later.

Importance of consistency

In our example, we'll build an events scheduler. If we are available for a meeting and we get two concurrent requests for a meeting invite, we'll get double-booked should a race condition exist. Alternately, locked data across two goroutines may cause both the requests to be denied or will result in an actual deadlock.

We want to guarantee that any request for availability is consistent — there should neither be double-booking nor should a request for an event be blocked incorrectly (because two concurrent or parallel routines lock the data simultaneously).

Synchronizing our concurrent operations

The word synchronization literally refers to temporal existence — things occurring at the same time. It seems then that the most apt demonstration of synchronicity will be something involving time itself.

When we think about the ways time impacts us, it's generally a matter of scheduling, due dates, and coordination. Going back to our preliminary example from the Preface, if one wishes to plan their grandmother's birthday party, the following types of scheduled tasks can take several forms:

- Things that must be done by a certain time (the actual party)
- Things that cannot be done until another task is completed (putting up decorations before they're purchased)
- Things that can be done in any particular order without impacting the outcome (cleaning the house)
- Things that can be done in any order but may well impact the outcome (buying a cake before finding out what cake your grandmother likes the most)

With these in mind, we'll attempt to handle some rudimentary human scheduling by designing an appointment calendar that can handle any number of people with one hour timeslots between 9 a.m. and 5 p.m.

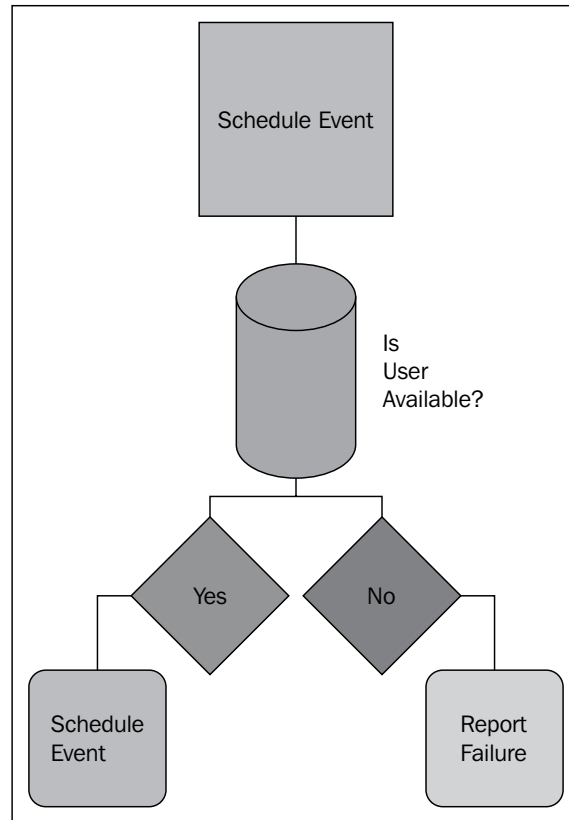
The project – multiuser appointment calendar

What do you do when you decide to write a program?

If you're like a lot of people, you think about the program; perhaps you and a team will write up a spec or requirements document, and then you'll get to coding. Sometimes, there will be a drawing representing some facsimile of the way the application will work.

Quite often, the best way to nail down the architecture and the inner workings of an application is to put pencil to paper and visually represent the way the program will work. For a lot of linear or serial applications, this is often an unnecessary step as things will work in a predictable fashion that should not require any specific coordination within the application logic itself (although coordinating third-party software likely benefits from specification).

You may be familiar with some logic that looks something like the following diagram:



The logic here makes sense. If you remember from our Preface, when humans draw out processes, we tend to serialize them. Visually, going from step one to step two with a finite number of processes is easy to understand.

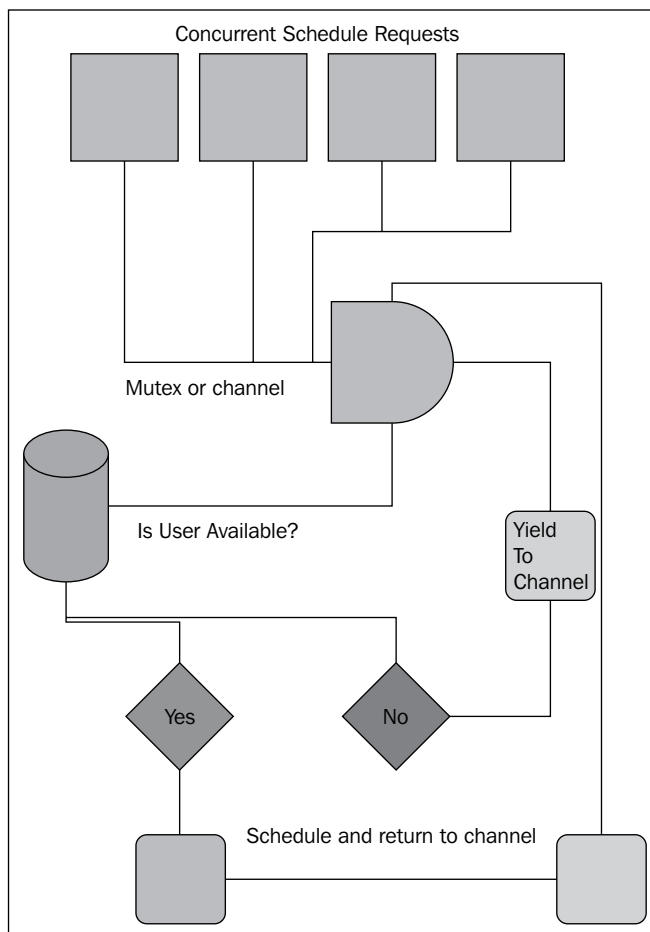
However, when designing a concurrent application, it's essential that we at least account for innumerable and concurrent requests, processes, and logic to make sure our application ends where we want, with the data and results we expect.

In the previous example, we completely ignore the possibility that "Is User Available" could fail or report old or erroneous data. Does it make more sense to address such problems if and when we find them, or should we anticipate them as part of a control flow? Adding complexity to the model can help us reduce the odds of data integrity issues down the road.

Let's visualize this again, taking into account availability pollers that will request availability for a user with any given request for a time/user pair.

Visualizing a concurrent pattern

As we have already discussed, we wish to create a basic blueprint of how our application should function as a starting point. Here, we'll implement some control flow, which relates to user activity, to help us decide what functionality we'll need to include. The following diagram illustrates how the control flow may look like:



In the previous diagram, we anticipate where data can be shared using concurrent and parallel processes to locate points of failure. If we design concurrent applications in such graphical ways, we're less likely to find race conditions later on.

While we talked about how Go helps you to locate these after the application has completed running, our ideal development workflow is to attempt to cut these problems off at the start.

Developing our server requirements

Now that we have an idea of how the scheduling process should work, we need to identify components that our application will need. In this case, the components are as follows:

- A web server handler
- A template for output
- A system for determining dates and times

Web server

In our visualizing concurrency example from the previous chapter, we used Go's built-in `http` package, and we'll do the same here. There are a number of good frameworks out there for this, but they primarily extend the core Go functionality rather than reinventing the wheel. The following are a few of these functionalities, listed from lightest to heaviest:

- Web.go: <http://webgo.io/>
Web.go is very lightweight and lean, and it provides some routing functionality not available in the `net/http` package.
- Gorilla: <http://www.gorillatoolkit.org/>
Gorilla is a Swiss army knife to augment the `net/http` package. It's not particularly heavy, and it is fast, utilitarian, and very clean.
- Revel: <http://robfig.github.io/revel/>
Revel is the heaviest of the three, but it focuses on a lot of intuitive code, caching, and performance. Look for it if you need something mature that will face a lot of traffic.

In *Chapter 6, C10K – A Non-blocking Web Server in Go*, we'll roll our own web server and framework with the sole goal of extreme high performance.

The Gorilla toolkit

For this application, we'll partially employ the Gorilla web toolkit. Gorilla is a fairly mature web-serving platform that fulfills a few of our needs here natively, namely the ability to include regular expressions in our URL routing. (Note: Web.Go also extends some of this functionality.) Go's internal HTTP routing handler is rather simplistic; you can extend this, of course, but we'll take a shortcut down a well-worn and reliable path here.

We'll use this package solely for ease of URL routing, but the Gorilla web toolkit also includes packages to handle cookies, sessions, and request variables. We'll examine this package a little closer in *Chapter 6, C10K – A Non-blocking Web Server in Go*.

Using templates

As Go is intended as a system language, and as system languages often deal with the creation of servers with clients, some care was put into making it a well-featured alternative to create web servers.

Anyone who's dealt with a "web language" will know that on top of that you'll need a framework, ideally one that handles the presentation layer for the web. While it's true that if you take on such a project you'll likely look for or build your own framework, Go makes the templating side of things very easy.

The template package comes in two varieties: `text` and `http`. Though they both serve different end points, the same properties—affording dynamism and flexibility—apply to the presentation layer rather than strictly the application layer.



The `text` template package is intended for general plaintext documents, while the `http` template package handles the generation of HTML and related documents.

These templating paradigms are all too common these days; if you look at the `http/template` package, you'll find some very strong similarities to Mustache, one of the more popular variants. While there is a Mustache port in Go, there's nothing there that isn't handled by default in the template package.



For more information on Mustache, visit <http://mustache.github.io/>.

One potential advantage to Mustache is its availability in other languages. If you ever feel the need to port some of your application logic to another language (or existing templates into Go), utilizing Mustache could be advantageous. That said, you sacrifice a lot of the extended functionality of Go templates, namely the ability to take out Go code from your compiled package and move it directly into template control structures. While Mustache (and its variants) has control flows, they may not mirror Go's templating system. Take the following example:

```
<ul>
  {{range .Users}}
  <li>A User </li>
  {{end}}
</ul>
```

Given the familiarity with Go's logic structures, it makes sense to keep them consistent in our templating language as well.



We won't show all the specific templates in this thread, but we will show the output. If you wish to peruse them, they're available at mastergoco.com/chapters/3/templates.

Time

We're not doing a whole lot of math here; time will be broken into hour blocks and each will be set to either occupied or available. At this time, there aren't a lot of external `date/time` packages for Go. We're not doing any heavy-date math, but it doesn't really matter because Go's `time` package should suffice even if we were.

In fact, as we have literal hour blocks from 9 a.m. to 5 p.m., we just set these to the 24-hour time values of 9-17, and invoke a function to translate them into linguistic dates.

Endpoints

We'll want to identify the REST endpoints (via `GET` requests) and briefly describe how they'll work. You can think of these as modules or methods in the model-view-controller architecture. The following is a list of the endpoint patterns we'll use:

- `entrypoint/register/{name}`: This is where we'll go to add a name to the list of users. If the user exists, it will fail.
- `entrypoint/viewusers`: Here, we'll present a list of users with their timeslots, both available and occupied.
- `entrypoint/schedule/{name}/{time}`: This will initialize an attempt to schedule an appointment.

Each will have an accompanying template that will report the status of the intended action.

Custom structs

We'll deal with users and responses (web pages), so we need two structs to represent each. One struct is as follows:

```
type User struct {
    Name string
    email string
    times[int] bool
}
```

The other struct is as follows:

```
type Page struct {
    Title string
    Body string
}
```

We will keep the page as simple as possible. Rather than doing a lot of iterative loops, we will produce the HTML within the code for the most part.

Our endpoints for requests will relate to our previous architecture, using the following code:

```
func users(w http.ResponseWriter, r *http.Request) {
}
func register(w http.ResponseWriter, r *http.Request) {
}
func schedule(w http.ResponseWriter, r *http.Request) {
}
```

A multiuser Appointments Calendar

In this section, we'll quickly look at our sample Appointments Calendar application, which attempts to control consistency of specific elements to avoid obvious race conditions. The following is the full code, including the routing and templating:

```
package main

import (
    "net/http"
    "html/template"
    "fmt"
```

```
    "github.com/gorilla/mux"
    "sync"
    "strconv"
)

type User struct {
    Name string
    Times map[int] bool
    DateHTML template.HTML
}

type Page struct {
    Title string
    Body template.HTML
    Users map[string] User
}

var usersInit map[string] bool
var userIndex int
var validTimes []int
var mutex sync.Mutex
var Users map[string]User
var templates = template.Must(template.New("template").
ParseFiles("view_users.html", "register.html"))

func register(w http.ResponseWriter, r *http.Request){
    fmt.Println("Request to /register")
    params := mux.Vars(r)
    name := params["name"]

    if _,ok := Users[name]; ok {
        t,_ := template.ParseFiles("generic.txt")
        page := &Page{ Title: "User already exists", Body:
            template.HTML("User " + name + " already exists")}
        t.Execute(w, page)
    } else {
        newUser := User { Name: name }
        initUser(&newUser)
        Users[name] = newUser
        t,_ := template.ParseFiles("generic.txt")
        page := &Page{ Title: "User created!", Body:
            template.HTML("You have created user "+name)}
    }
}
```

```

        t.Execute(w, page)
    }

}

func dismissData(st1 int, st2 bool) {

// Does nothing in particular for now other than avoid Go compiler
errors
}

func formatTime(hour int) string {
    hourText := hour
    ampm := "am"
    if (hour > 11) {
        ampm = "pm"
    }
    if (hour > 12) {
        hourText = hour - 12;
    }
    fmt.Println(ampm)
    outputString := strconv.FormatInt(int64(hourText),10) + ampm

    return outputString
}

func (u User) FormatAvailableTimes() template.HTML { HTML := ""
    HTML += "<b>" + u.Name + "</b> - "

    for k,v := range u.Times { dismissData(k,v)

        if (u.Times[k] == true) { formattedTime := formatTime(k) HTML
            += "<a href='/schedule/' + u.Name + '/' +
                + strconv.FormatInt(int64(k),10) + '"
                class='button'>" + formattedTime + "</a> "

        } else {

        }

    } return template.HTML(HTML)
}

```

```
}

func users(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Request to /users")

    t,_ := template.ParseFiles("users.txt")
    page := &Page{ Title: "View Users", Users: Users}
    t.Execute(w, page)
}

func schedule(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Request to /schedule")
    params := mux.Vars(r)
    name := params["name"]
    time := params["hour"]
    timeVal,_ := strconv.ParseInt( time, 10, 0 )
    intTimeVal := int(timeVal)

    createURL := "/register/"+name

    if _,ok := Users[name]; ok {
        if Users[name].Times[intTimeVal] == true {
            mutex.Lock()
            Users[name].Times[intTimeVal] = false
            mutex.Unlock()
            fmt.Println("User exists, variable should be modified")
            t,_ := template.ParseFiles("generic.txt")
            page := &Page{ Title: "Successfully Scheduled!", Body:
                template.HTML("This appointment has been scheduled. <a
                    href='/users'>Back to users</a>")}

            t.Execute(w, page)

        } else {
            fmt.Println("User exists, spot is taken!")
            t,_ := template.ParseFiles("generic.txt")
            page := &Page{ Title: "Booked!", Body:
                template.HTML("Sorry, "+name+" is booked for
                    "+time+" <a href='/users'>Back to users</a>")}
            t.Execute(w, page)
        }
    }
}
```

```

    } else {
        fmt.Println("User does not exist")
        t,_ := template.ParseFiles("generic.txt")
        page := &Page{ Title: "User Does Not Exist!", Body:
            template.HTML( "Sorry, that user does not exist. Click
                <a href='"+createUrl+"'>here</a> to create it. <a
                    href='/users'>Back to users</a>") }
        t.Execute(w, page)
    }
    fmt.Println(name,time)
}

func defaultPage(w http.ResponseWriter, r *http.Request) {

}

func initUser(user *User) {

    user.Times = make(map[int] bool)
    for i := 9; i < 18; i ++ {
        user.Times[i] = true
    }

}

func main() {
    Users = make(map[string] User)
    userIndex = 0
    bill := User {Name: "Bill" }
    initUser(&bill)
    Users["Bill"] = bill
    userIndex++

    r := mux.NewRouter()  r.HandleFunc("/", defaultPage)
    r.HandleFunc("/users", users)
    r.HandleFunc("/register/{name:[A-Za-z]+}", register)
    r.HandleFunc("/schedule/{name:[A-Za-z]+}/{hour:[0-9]+}",
        schedule)        http.Handle("/", r)

    err := http.ListenAndServe(":1900", nil)  if err != nil {    //
        log.Fatal("ListenAndServe:", err)    }

}

```


Note that we seeded our application with a user, Bill. If you attempt to hit `/register/bill|bill@example.com`, the application will report that the user exists.

As we control the most sensitive data through channels, we avoid any race conditions. We can test this in a couple of ways. The first and easiest way is to keep a log of how many successful appointments are registered, and run this with Bill as the default user.

We can then run a concurrent load tester against the action. There are a number of such testers available, including Apache's `ab` and `Siege`. For our purposes, we'll use `JMeter`, primarily because it permits us to test against multiple URLs concurrently.

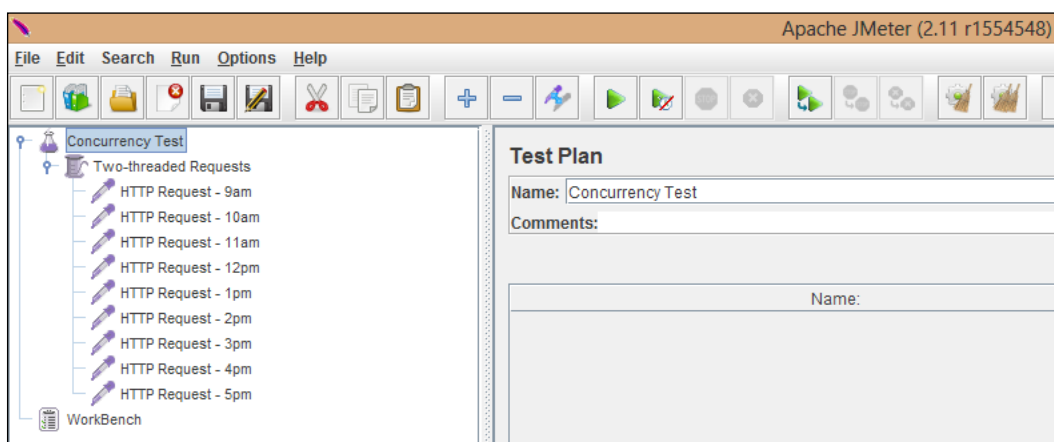


Although we're not necessarily using `JMeter` for load testing (rather, we use it to run concurrent tests), load testers can be extraordinarily valuable ways to find bottlenecks in applications at scales that don't yet exist.

For example, if you built a web application that had a blocking element and had 5,000-10,000 requests per day, you may not notice it. But at 5 million-10 million requests per day, it might result in the application crashing.

In the dawn of network servers, this is what happened; servers scaled until one day, suddenly, they couldn't scale further. Load/stress testers allow you to simulate traffic in order to better detect these issues and inefficiencies.

Given that we have one user and eight hours in a day, we should end our script with no more than eight total successful appointments. Of course, if you hit the `/register` endpoint, you will see eight times as many users as you've added. The following screenshot shows our benchmark test plan in `JMeter`:

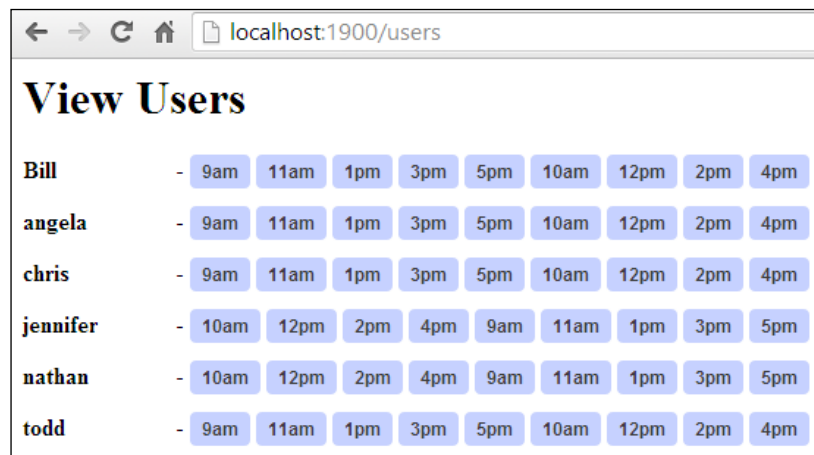


When you run your application, keep an eye on your console; at the end of our load test, we should see the following message:

Total registered appointments: 8

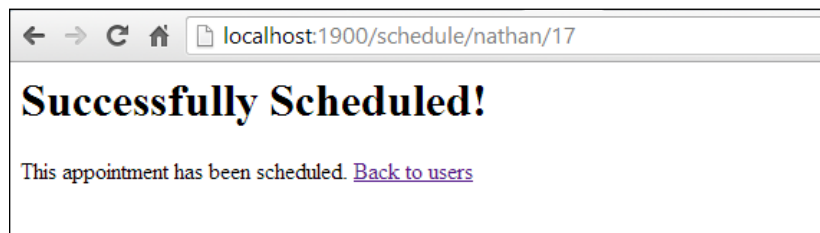
Had we designed our application as per the initial graphical mockup representation in this chapter (with race conditions), it's plausible – and in fact likely – that we'd register far more appointments than actually existed.

By isolating potential race conditions, we guarantee data consistency and ensure that nobody is waiting on an appointment with an otherwise occupied attendee. The following screenshot is the list we present of all the users and their available appointment times:

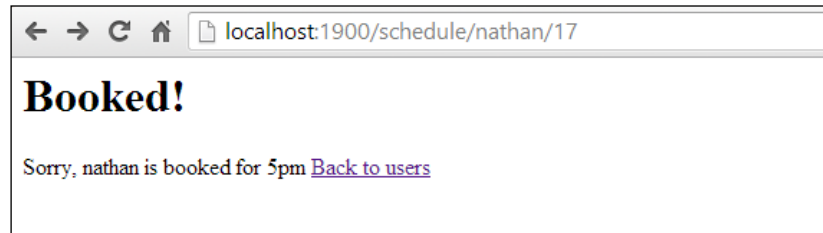


The previous screenshot is our initial view that shows us available users and their available time slots. By selecting a timeslot for a user, we'll attempt to book them for that particular time. We'll start with Nathan at 5 p.m.

The following screenshot shows what happens when we attempt to schedule with an available user:



However, if we attempt to book again (even simultaneously), we'll be greeted with a sad message that Nathan cannot see us at 5 p.m, as shown in the following screenshot:



With that, we have a multiuser calendar app that allows for creating new users, scheduling, and blocking double-bookings.

Let's look at a few interesting new points in this application.

First, you will notice that we use a template called `generic.txt` for most parts of the application. There's not much to this, only a page title and body filled in by each handler. However, on the `/users` endpoint, we use `users.txt` as follows:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-
    8">
  <title>{{.Title}}</title>
</head>
<body>

<h1>{{.Title}}</h1>

{{range .Users}}
<div class="user-row">

  {{.FormatAvailableTimes}}

</div>
{{end}}

</body>
</html>
```

We mentioned the range-based functionality in templates, but how does `{{.FormatAvailableTimes}}` work? In any given context, we can have type-specific functions that process the data in more complex ways than are available strictly in the template lexer.

In this case, the `User` struct is passed to the following line of code:

```
func (u User) FormatAvailableTimes() template.HTML {
```

This line of code then performs some conditional analysis and returns a string with some time conversion.

In this example, you can use either a channel to control the flow of `User.times` or an explicit mutex as we have here. We don't want to limit all locks, unless absolutely necessary, so we only invoke the `Lock()` function if we've determined the request has passed the tests necessary to modify the status of any given user/time pair. The following code shows where we set the availability of a user within a mutual exclusion:

```
if _,ok := Users[name]; ok {
    if Users[name].Times[intTimeVal] == true {
        mutex.Lock()
        Users[name].Times[intTimeVal] = false
        mutex.Unlock()
    }
}
```

The outer evaluation checks that a user by that name (key) exists. The second evaluation checks that the time availability exists (true). If it does, we lock the variable, set it to false, and then move onto output rendering.

Without the `Lock()` function, many concurrent connections can compromise the consistency of data and cause the user to have more than one appointment in a given hour.

A note on style

You'll note that despite preferring camelCase for most of our variables, we have some uppercase variables within structs. This is an important Go convention worth mentioning: any struct variable that begins with a capital letter is **public**. Any variable that begins with a lowercase letter is **private**.

If you attempt to output a private (or nonexistent) variable in your template files, template rendering will fail.

A note on immutability

Note that whenever possible, we'll avoid using the string type for comparative operations, especially in multithreaded environments. In the previous example, we use integers and Booleans to decide availability for any given user. In some languages, you may feel empowered to assign the time values to a string for ease of use. For the most part, this is fine, even in Go; but assuming that we have an infinitely scalable, shared calendar application, we run the risk of introducing memory issues if we utilize strings in this way.

The string type is the sole immutable type in Go; this is noteworthy if you end up assigning and reassigning values to a string. Assuming that memory is yielded after a string is converted to a copy, this is not a problem. However, in Go (and a couple of other languages), it's entirely possible to keep the original value in memory. We can test this using the following example:

```
func main() {  
  
    testString := "Watch your top / resource monitor"  
    for i:= 0; i < 1000; i++ {  
  
        testString = string(i)  
  
    }  
    doNothing(testString)  
  
    time.Sleep(10 * time.Second)  
  
}
```

When run in Ubuntu, this takes approximately 1.0 MB of memory; some of that no doubt overhead, but a useful reference point. Let's up the ante a bit – though having 1,000 relatively small pointers won't have much impact – using the following line of code:

```
for i:= 0; i < 100000000; i++ {
```

Now, having gone through 100 million memory assignments, you can see the impact on memory (it doesn't help that the string itself is at this point longer than the initial, but it doesn't account for the full effect). Garbage collection takes place here too, which impacts CPU. On our initial test here, both CPU and memory spiked. If we substitute this for an integer or a Boolean assignment, we get much smaller footprints.

This is not exactly a real-world scenario, but it's worth noting in a concurrent environment where garbage collection must happen so we can evaluate the properties and types of our logic.

It's also entirely possible, depending on your current version of Go, your machine(s), and so on, and this could run as efficiently in either scenario. While that might seem fine, part of our concurrent strategy planning should involve the possibility that our application will scale in input, output, physical resources, or all of them. Just because something works well now doesn't mean it's not worth implementing efficiencies that will keep it from causing performance problems at a 100x scale.

If you ever encounter a place where a string is logical, but you want or could benefit from a mutable type, consider a byte slice instead.

A constant is, of course, also immutable, but given that's the implied purpose of a constant variable, you should already know this. A mutable constant variable is, after all, an oxymoron.

Summary

This chapter has hopefully directed you towards exploring methods to plan and chart out your concurrent applications before delving in. By briefly touching on race conditions and data consistency, we attempted to highlight the importance of anticipatory design. At the same time, we utilized a few tools for identifying such issues, should they occur.

Creating a robust script flowchart with concurrent processes will help you locate possible pitfalls before you create them, and it will give you a better sense of how (and when) your application should be making decisions with logic and data.

In the next chapter, we'll examine data consistency issues and look at advanced channel communication options in an effort to avoid needless and often expensive mitigating functions, mutexes, and external processes.

4

Data Integrity in an Application

By now, you should be comfortable with the models and tools provided in Go's core to provide mostly race-free concurrency.

We can now create goroutines and channels with ease, manage basic communication across channels, coordinate data without race conditions, and detect such conditions as they arise.

However, we can neither manage larger distributed systems nor deal with potentially lower-level consistency problems. We've utilized a basic and simplistic mutex, but we are about to look at a more complicated and expressive way of handling mutual exclusions.

By the end of this chapter, you should be able to expand your concurrency patterns from the previous chapter into distributed systems using a myriad of concurrency models and systems from other languages. We'll also look—at a high level—at some consistency models that you can utilize to further express your precoding strategies for single-source and distributed applications.

Getting deeper with mutexes and sync

In *Chapter 2, Understanding the Concurrency Model*, we introduced `sync.Mutex` and how to invoke a mutual exclusion lock within your code, but there's some more nuance to consider with the package and the mutex type.

We've mentioned that in an ideal world, you should be able to maintain synchronization in your application by using goroutines alone. In fact, this would probably be best described as the canonical method within Go, although the `sync` package does provide a few other utilities, including mutexes.

Whenever possible, we'll stick with goroutines and channels to manage consistency, but the mutex does provide a more traditional and granular approach to lock and access data. If you've ever managed another concurrent language (or package within a language), odds are you've had experience with either a mutex or a philosophical analog. In the following chapters, we'll look at ways of extending and exploiting mutexes to do a little more out of the box.

If we look at the `sync` package, we'll see there are a couple of different mutex structs.

The first is `sync.Mutex`, which we've explored – but another is `RWMutex`. The `RWMutex` struct provides a multireader, single-writer lock. These can be useful if you want to allow reads to resources but provide mutex-like locks when a write is attempted. They can be best utilized when you expect a function or subprocess to do frequent reads but infrequent writes, but it still cannot afford a dirty read.

Let's look at an example that updates the date/time every 10 seconds (acquiring a lock), yet outputs the current value every other second, as shown in the following code:

```
package main

import (
    "fmt"
    "sync"
    "time"
)

type TimeStruct struct {
    totalChanges int
    currentTime time.Time
    rwLock sync.RWMutex
}

var TimeElement TimeStruct

func updateTime() {
    TimeElement.rwLock.Lock()
    defer TimeElement.rwLock.Unlock()
    TimeElement.currentTime = time.Now()
    TimeElement.totalChanges++
}

func main() {

    var wg sync.WaitGroup
```

```

TimeElement.totalChanges = 0
TimeElement.currentTime = time.Now()
timer := time.NewTicker(1 * time.Second)
writeTimer := time.NewTicker(10 * time.Second)
endTimer := make(chan bool)

wg.Add(1)
go func() {

    for {
        select {
            case <-timer.C:
                fmt.Println(TimeElement.totalChanges,
                    TimeElement.currentTime.String())
            case <-writeTimer.C:
                updateTime()
            case <-endTimer:
                timer.Stop()
                return
        }
    }

}()

wg.Wait()
fmt.Println(TimeElement.currentTime.String())
}

```



We don't explicitly run `Done()` on our `WaitGroup` struct, so this will run in perpetuity.

There are two different methods for performing locks/unlocks on `RWMutex`:

- `Lock()`: This will block variables for both reading and writing until an `Unlock()` method is called
- `happenedRlock()`: This locks bound variables solely for reads

The second method is what we've used for this example, because we want to simulate a real-world lock. The net effect is the `interval` function that outputs the current time that will return a single dirty read before `rwLock` releases the read lock on the `currentTime` variable. The `Sleep()` method exists solely to give us time to witness the lock in motion. An `RWLock` struct can be acquired by many readers or by a single writer.

The cost of goroutines

As you work with goroutines, you might get to a point where you're spawning dozens or even hundreds of them and wonder if this is going to be expensive. This is particularly true if your previous experience with concurrent and/or parallel programming was primarily thread-based. It's commonly accepted that maintaining threads and their respective stacks can begin to bog down a program with performance issues. There are a few reasons for this, which are as follows:

- Memory is required just for the creation of a thread
- Context switching at the OS level is more complex and expensive than in-process context switching
- Very often, a thread is spawned for a very small process that could be handled otherwise

It's for these reasons that a lot of modern concurrent languages implement something akin to goroutines (C# uses the `async` and `await` mechanism, Python has `greenlets`/`green threads`, and so on) that simulate threads using small-scale context switching.

However, it's worth knowing that while goroutines are (or can be) cheap and cheaper than OS threads, they are not free. At a large (perhaps enormous) measure, even cheap and light goroutines can impact performance. This is particularly important to note as we begin to look at distributed systems, which often scale larger and at faster rates.

The difference between running a function directly and running it in a goroutine is negligible of course. However, keep in mind that Go's documentation states:

It is practical to create hundreds of thousands of goroutines in the same address space.

Given that stack creation uses a few kilobytes per goroutine, in a modern environment, it's easy to see how that could be perceived as a nonfactor. However, when you start talking about thousands (or millions) of goroutines running, it can and likely will impact the performance of any given subprocess or function. You can test this by wrapping functions in an arbitrary number of goroutines and benchmarking the average execution time and – more importantly – memory usage. At approximately 5KB per goroutine, you may find that memory can become a factor, particularly on low-RAM machines or instances. If you have an application that runs heavy on a high-powered machine, imagine it reaching criticality in one or more lower-powered machines. Consider the following example:

```
for i:= 0; i < 1000000000; i++ {  
    go someFunction()  
}
```

Even if the overhead for the goroutine is cheap, what happens at 100 million or — as we have here — a billion goroutines running?

As always, doing this in an environment that utilizes more than a single core can actually increase the overhead of this application due to the costs of OS threading and subsequent context switching.

These issues are almost always the ones that are invisible unless and until an application begins to scale. Running on your machine is one thing, running at scale across a distributed system with what amounts to low-powered application servers is quite another.

The relationship between performance and data consistency is important, particularly if you start utilizing a lot of goroutines with mutual exclusions, locks, or channel communication.

This becomes a larger issue when dealing with external, more permanent memory sources.

Working with files

Files are a great example of areas where data consistency issues such as race conditions can lead to more permanent and catastrophic problems. Let's look at a piece of code that might continuously attempt to update a file to see where we could run into race conditions, which in turn could lead to bigger problems such as an application failing or losing data consistency:

```
package main

import (
    "fmt"
    "io/ioutil"
    "strconv"
    "sync"
)

func writeFile(i int) {

    rwLock.RLock();
    ioutil.WriteFile("test.txt",
        []byte(strconv.FormatInt(int64(i), 10)), 0x777)
    rwLock.RUnlock();
}
```

```
writer<-true

}

var writer chan bool
var rwLock sync.RWMutex

func main() {

    writer = make(chan bool)

    for i:=0;i<10;i++ {
        go writeFile(i)
    }

    <-writer
    fmt.Println("Done!")
}
```

Code involving file operations are rife for these sorts of potential issues, as mistakes are specifically *not ephemeral* and can be locked in time forever.

If our goroutines block at some critical point or the application fails midway through, we could end up with a file that has invalid data in it. In this case, we're simply iterating through some numbers, but you can also apply this situation to one involving database or datastore writes – the potential exists for persistent bad data instead of temporary bad data.

This is not a problem that is exclusively solved by channels or mutual exclusions; rather, it requires some sort of sanity check at every step to make certain that data is where you and the application expect it to be at every step in the execution. Any operation involving `io.Writer` relies on primitives, which Go's documentation explicitly notes that we should not assume they are safe for parallel execution. In this case, we have wrapped the file writing in a mutex.

Getting low – implementing C

One of the most interesting developments in language design in the past decade or two is the desire to implement lower-level languages and language features via API. Java lets you do this purely externally, and Python provides a C library for interaction between the languages. It warrants mentioning that the reasons for doing this vary – among them applying Go's concurrency features as a wrapper for legacy C code – and you will likely have to deal with some of the memory management associated with introducing unmanaged code to garbage-collected applications.

Go takes a hybrid approach, allowing you to call a C interface through an import, which requires a frontend compiler such as GCC:

```
import "C"
```

So why would we want to do this?

There are some good and bad reasons to implement C directly in your project. An example of a good reason might be to have direct access to the inline assembly, which you can do in C but not directly in Go. A bad reason could be any that has a solution inherent in Golang itself.

To be fair, even a bad reason is not bad if you build your application reliably, but it does impose an additional level of complexity to anyone else who might use your code. If Go can satisfy the technical and performance requirements, it's always better to use a single language in a single project.

There's a famous quote from C++ creator Bjarne Stroustrup on C and C++:

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows your whole leg off.

Jokes aside (Stroustrup has a vast collection of such quips and quotes), the fundamental reasoning is that the complexity of C often prevents people from accidentally doing something catastrophic.

As Stroustrup says, C makes it easy to make big mistakes, but the repercussions are often smaller due to language design than higher-level languages. Issues dealing with security and stability are easy to be introduced in any low-level language.

By simplifying the language, C++ provides abstractions that make low-level operations easier to carry out. You can see how this might apply to using C directly in Go, given the latter language's syntactical sweetness and programmer friendliness.

That said, working with C can highlight some of the potential pitfalls with regard to memory, pointers, deadlocks, and consistency, so we'll touch upon a simple example as follows:

```
package main

// #include <stdio.h>
// #include <string.h>
// int string_length (char* str) {
//     return strlen(str);
// }
import "C"
import "fmt"
func main() {
    v := C.CString("Don't Forget My Memory Is Not Visible To Go!")
    x := C.string_length(v)
    fmt.Println("A C function has determined your string
        is", x, "characters in length")
}
```

Touching memory in cgo

The most important takeaway from the preceding example is to remember that anytime you go into or out of C, you need to manage memory manually (or at least more directly than with Go alone). If you've ever worked in C (or C++), you know that there's no automatic garbage collection, so if you request memory space, you must also free it. Calling C from Go does not preclude this.

The structure of cgo

Importing C into Go will take you down a syntactical side route, as you probably noticed in the preceding code. The first thing that will appear glaringly different is the actual implementation of C code within your application.

Any code (in comments to stop Go's compiler from failing) directly above the `import "C"` directive will be interpreted as C code. The following is an example of a C function declared above our Go code:

```
/*
    int addition(int a, int b) {
        return a + b;
    }

```

Bear in mind that Go won't validate this, so if you make an error in your C code, it could lead to silent failure.

Another related warning is to remember your syntax. While Go and C share a lot of syntactical overlap, leave off a curly bracket or a semicolon and you could very well find yourself in one of those silent failure situations. Alternately, if you're working in the C part of your application and you go back to Go, you will undoubtedly find yourself wrapping loop expressions in parentheses and ending your lines with semicolons.

Also remember that you'll frequently have to handle type conversions between C and Go that don't have one-to-one analogs. For example, C does not have a built-in string type (you can, of course, include additional libraries for types), so you may need to convert between strings and char arrays. Similarly, `int` and `int64` might need some nonimplicit conversion, and again, you may not get the debugging feedback that you might expect when compiling these.

The other way around

Using C within Go is obviously a potentially powerful tool for code migration, implementing lower-level code, and roping in other developers, but what about the inverse? Just as you can call C from within Go, you can call Go functions as external functions within your embedded C.

The end game here is the ability to work with and within C and Go in the same application. By far the easiest way to handle this is by using `gccgo`, which is a frontend for GCC. This is different than the built-in Go compiler; it is possible to go back and forth between C and Go without `gccgo`, but using it makes this process much simpler.

gopart.go

The following is the code for the Go part of the interaction, which the C part will call as an external function:

```
package main

func MyGoFunction(num C.int) int {

    squared := num * num
    fmt.Println(num,"squared is",squared)
    return squared
}
```

cpart.c

Now for the C part, where we make our call to our Go application's exported function `MyGoFunction`, as shown in the following code snippet:

```
#include <stdio.h>

extern int square_it(int) __asm__ ("cross.main.MyGoFunction")

int main() {

    int output = square_it(5)
    printf("Output: %d",output)
    return 0;
}
```

Makefile

Unlike using C directly in Go, at present, doing the inverse requires the use of a makefile for C compilation. Here's one that you can use to get an executable from the earlier simple example:

```
all: main

main: cpart.o cpart.c
    gcc cpart.o cpart.c -o main

gopart.o: gopart.go
    gccgo -c gopart.go -o gopart.o -fgo-prefix=cross

clean:
    rm -f main *.o
```

Running the makefile here should produce an executable file that calls the function from within C.

However, more fundamentally, `cgo` allows you to define your functions as external functions for C directly:

```
package output

import "C"

//export MyGoFunction
func MyGoFunction(num int) int {

    squared := num * num
    return squared
}
```

Next, you'll need to use the `cgo` tool directly to generate header files for C as shown in the following line of code:

```
go tool cgo goback.go
```

At this point, the Go function is available for use in your C application:

```
#include <stdio.h>
#include "_obj/_cgo_export.h"

extern int MyGoFunction(int num);

int main() {

    int result = MyGoFunction(5);
    printf("Output: %d", result);
    return 0;

}
```

Note that if you export a Go function that contains more than one return value, it will be available as a struct in C rather than a function, as C does not provide multiple variables returned from a function.

At this point, you may be realizing that the true power of this functionality is the ability to interface with a Go application directly from existing C (or even C++) applications.

While not necessarily a true API, you can now treat Go applications as linked libraries within C apps and vice versa.

One caveat about using `//export` directives: if you do this, your C code must reference these as extern-declared functions. As you may know, `extern` is used when a C application needs to call a function from another linked C file.

When we build our Go code in this manner, `cgo` generates the header file `_cgo_export.h`, as you saw earlier. If you want to take a look at that code, it can help you understand how Go translates compiled applications into C header files for this type of use:

```
/* Created by cgo - DO NOT EDIT. */
#include "_cgo_export.h"

extern void crosscall2(void (*fn)(void *, int), void *, int);

extern void _cgoexp_d133c8d0d35b_MyGoFunction(void *, int);

GoInt64 MyGoFunction(GoInt p0)
{
    struct {
        GoInt p0;
        GoInt64 r0;
    } __attribute__((packed)) a;
    a.p0 = p0;
    crosscall2(_cgoexp_d133c8d0d35b_MyGoFunction, &a, 16);
    return a.r0;
}
```

You may also run into a rare scenario wherein the C code is not exactly as you expect, and you're unable to cajole the compiler to produce what you expect. In that case, you're always free to modify the header file before the compilation of your C application, despite the `DO NOT EDIT` warning.

Getting even lower – assembly in Go

If you can shoot your foot off with C and you can blow your leg off with C++, just imagine what you can do with assembly in Go.

It isn't possible to use assembly directly in Go, but as Go provides access to C directly and C provides the ability to call inline assembly, you can indirectly use it in Go.

But again, just because something is possible doesn't mean it should be done – if you find yourself in need of assembly in Go, you should consider using assembly directly and connecting via an API.

Among the many roadblocks that you may encounter with assembly in (C and then in) Go is the lack of portability. Writing inline C is one thing – your code should be relatively transferable between processor instruction sets and operating systems – but assembly is obviously something that requires a lot of specificity.

All that said, it's certainly better to have the option to shoot yourself in the foot whether you choose to take the shot or not. Use great care when considering whether you need C or assembly directly in your Go application. If you can get away with communicating between dissonant processes through an API or interprocess conduit, always take that route first.

One very obvious drawback of using assembly in Go (or on its own or in C) is you lose the cross-compilation capabilities that Go provides, so you'd have to modify your code for every destination CPU architecture. For this reason, the only practical times to use Go in C are when there is a single platform on which your application should run.

Here's an example of what an ASM-in-C-in-Go application might look like. Keep in mind that we've included no ASM code, because it varies from one processor type to another. Experiment with some boilerplate assembly in the following `__asm__` section:

```
package main

/*
#include <stdio.h>

void asmCall() {

__asm__( " " );
printf("I come from a %s", "C function with embedded asm\n");
```

```
}
*/
import "C"

func main() {

    C.asmCall()

}
```

If nothing else, this may provide an avenue for delving deeper into ASM even if you're familiar with neither assembly nor C itself. The more high-level you consider C and Go to be, the more practical you might see this.

For most uses, Go (and certainly C) is low-level enough to be able to squeeze out any performance hiccups without landing at assembly. It's worth noting again that while you do lose some immediate control of memory and pointers in Go when you invoke C applications, that caveat applies tenfold with assembly. All of those nifty tools that Go provides may not work reliably or not work at all. If you think about the Go race detector, consider the following application:

```
package main

/*
int increment(int i) {
    i++;
    return i;
}
*/
import "C"
import "fmt"

var myNumber int

func main() {
    fmt.Println(myNumber)

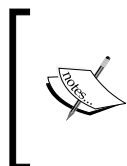
    for i:=0;i<100;i++ {
        myNumber = int( C.increment(C.int(myNumber)) )
        fmt.Println(myNumber)
    }

}
```

You can see how tossing your pointers around between Go and C might leave you out in the dark when you don't get what you expect out of the program.

Keep in mind that here there is a somewhat unique and perhaps unexpected kicker to using goroutines with cgo; they are treated by default as blocking. This isn't to say that you can't manage concurrency within C, but it won't happen by default. Instead, Go may well launch another system thread. You can manage this to some degree by utilizing the runtime function `runtime.LockOSThread()`. Using `LockOSThread` tells Go that a particular goroutine should stay within the present thread and no other concurrent goroutine may use this thread until `runtime.UnlockOSThread()` is called.

The usefulness of this depends heavily on the necessity to call C or a C library directly; some libraries will play happily as new threads are created, a few others may segfault.



Another useful runtime call you should find useful within your Go code is `NumGcCall()`. This returns the number of cgo calls made by a current process. If you need to lock and unlock threads, you can also use this to build an internal queue report to detect and prevent deadlocks.

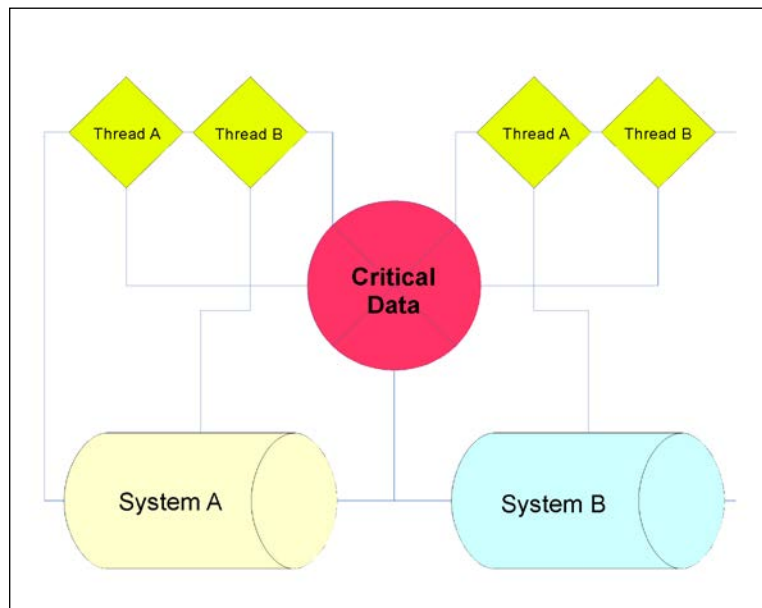
None of this precludes the possibility of race conditions should you choose to mix and match Go and C within goroutines.

Of course, C itself has a few race detector tools available. Go's race detector itself is based on the `ThreadSanitizer` library. It should go without saying that you probably do not want several tools that accomplish the same thing within a single project.

Distributed Go

So far, we've talked quite a bit about managing data within single machines, though with one or more cores. This is complicated enough as is. Preventing race conditions and deadlocks can be hard to begin with, but what happens when you introduce more machines (virtual or real) to the mix?

The first thing that should come to mind is that you can throw out a lot of the inherent tools that Go provides, and to a large degree that's true. You can mostly guarantee that Go can handle internal locking and unlocking of data within its own, singular goroutines and channels, but what about one or more additional instances of an application running? Consider the following model:



Here we see that either of these threads across either process could be reading from or writing to our **Critical Data** at any given point. With that in mind, there exists a need to coordinate access to that data.

At a very high level, there are two direct strategies for handling this, a distributed lock or consistency hash table (consistent hashing).

The first strategy is an extension of mutual exclusions except that we do not have direct and shared access to the same address space, so we need to create an abstraction. In other words, it's our job to concoct a lock mechanism that's visible to all available external entities.

The second strategy is a pattern designed specifically for caching and cache validation/invalidation, but it has relevancy here as well, because you can use it to manage where data lives in the more global address space.

However, when it comes to ensuring consistency across these systems, we need to go deeper than this general, high-level approach.

Split this model down the middle and it becomes easy: channels will handle the concurrent flow of data and data structures, and where they don't, you can use mutexes or low-level atomicity to add additional safeguards.

However, look to the right. Now you have another VM/instance or machine attempting to work with the same data. How can we make sure that we do not encounter reader/writer problems?

Some common consistency models

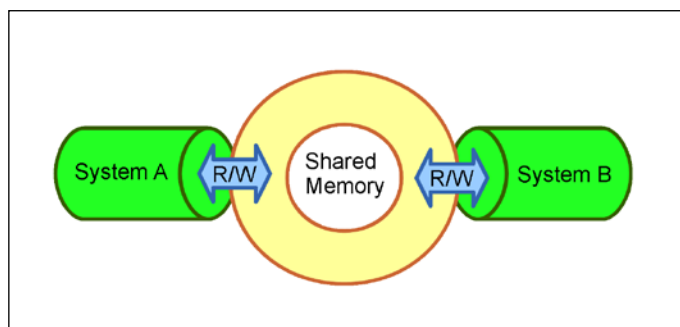
Luckily, there are some non-core Go solutions and strategies that we can utilize to improve our ability to control data consistency.

Let's briefly look at a few consistency models that we can employ to manage our data in distributed systems.

Distributed shared memory

On its own, a **Distributed Shared Memory (DSM)** system does not intrinsically prevent race conditions, as it is merely a method for more than one system to share real or partitioned memory.

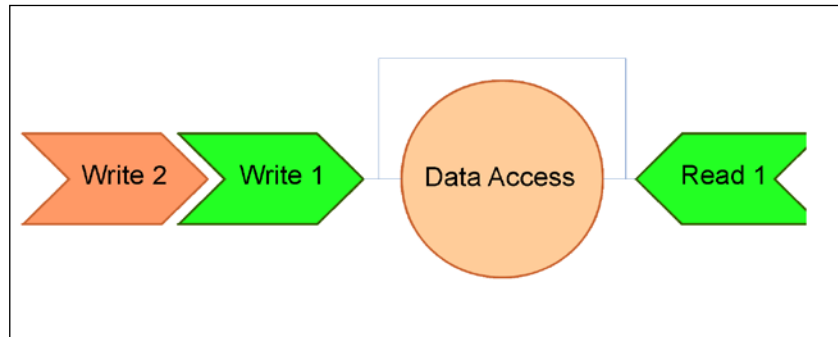
In essence, you can imagine two systems with 1 GB of memory, each allocating 500 MB to a shared memory space that is accessible and writable by each. Dirty reads are possible as are race conditions unless explicitly designed. The following figure is a visual representation of how two systems can coordinate using shared memory:



We'll look at one prolific but simple example of DSM shortly, and play with a library available to Go for test driving it.

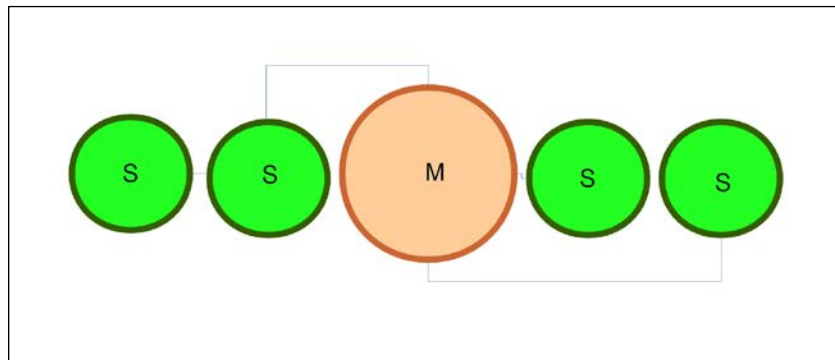
First-in-first-out – PRAM

Pipelined RAM (PRAM) consistency is a form of first-in-first-out methodology, in which data can be read in order of the queued writes. This means that writes read by any given, separate process may be different. The following figure represents this concept:



Looking at the master-slave model

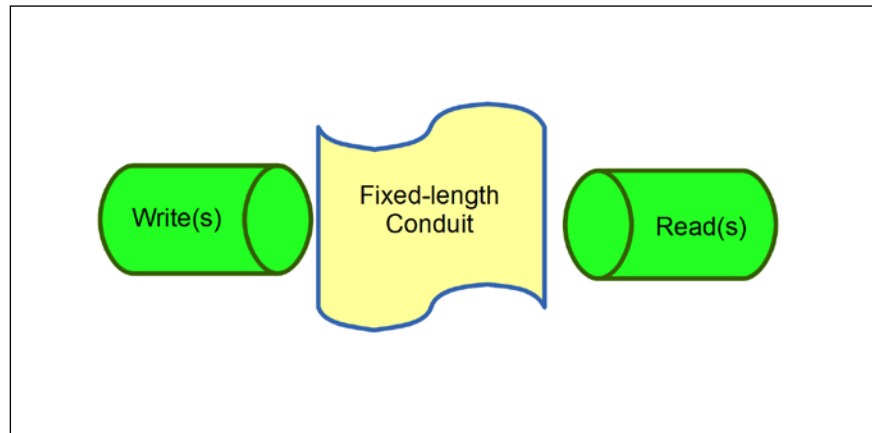
The master-slave consistency model is similar to the leader/follower model that we'll look at shortly, except that the master manages all operations on data and broadcasts rather than receiving write operations from a slave. In this case, replication is the primary method of transmission of changes to data from the master to the slave. In the following diagram, you will find a representation of the master-slave model with a master server and four slaves:



While we can simply duplicate this model in Go, we have more elegant solutions available to us.

The producer-consumer problem

In the classic producer-consumer problem, the producer writes chunks of data to a conduit/buffer, while a consumer reads chunks. The issue arises when the buffer is full: if the producer adds to the stack, the data read will not be what you intend. To avoid this, we employ a channel with waits and signals. This model looks a bit like the following figure:



If you're looking for the semaphore implementation in Go, there is no explicit usage of the semaphore. However, think about the language here – fixed-size channels with waits and signals; sounds like a buffered channel. Indeed, by providing a buffered channel in Go, you give the conduit here an explicit length; the channel mechanism gives you the communication for waits and signals. This is incorporated in Go's concurrency model. Let's take a quick look at a producer-consumer model as shown in the following code:

```
package main

import (
    "fmt"
)

var comm = make(chan bool)
var done = make(chan bool)

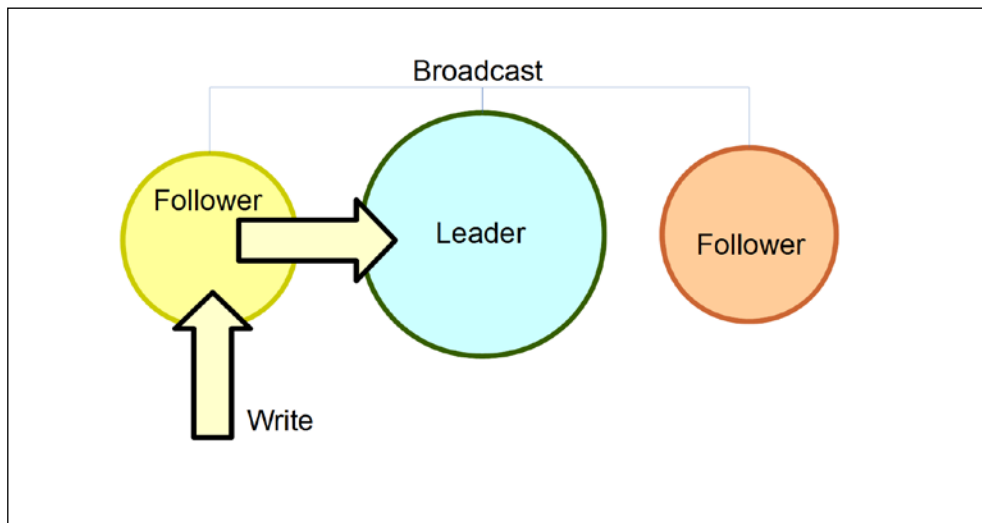
func producer() {
    for i:=0; i< 10; i++ {
        comm <- true
    }
}
```

```
done <- true
}
func consumer() {
  for {
    communication := <-comm
    fmt.Println("Communication from producer
      received!",communication)
  }
}

func main() {
  go producer()
  go consumer()
  <- done
  fmt.Println("All Done!")
}
```

Looking at the leader-follower model

In the leader/follower model, writes are broadcasted from a single source to any followers. Writes can be passed through any number of followers or be restricted to a single follower. Any completed writes are then broadcasted to the followers. This can be visually represented as the following figure:



We can see a channel analog here in Go as well. We can, and have, utilized a single channel to handle broadcasts to and from other followers.

Atomic consistency / mutual exclusion

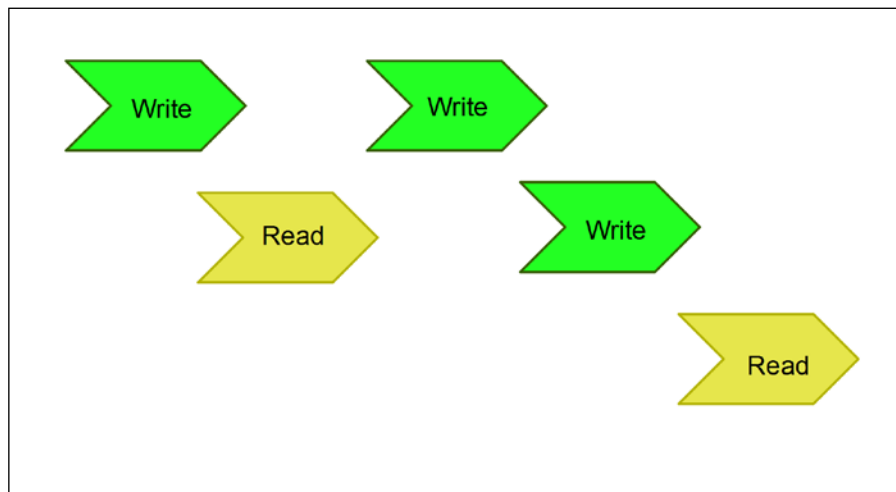
We've looked at atomic consistency quite a bit. It ensures that anything that is not created and used at essentially the same time will require serialization to guarantee the strongest form of consistency. If a value or dataset is not atomic in nature, we can always use a mutex to force linearizability on that data.

Serial or sequential consistency is inherently strong, but can also lead to performance issues and degradation of concurrency.

Atomic consistency is often considered the strongest form of ensuring consistency.

Release consistency

The release consistency model is a DSM variant that can delay a write's modifications until the time of first acquisition from a reader. This is known as lazy release consistency. We can visualize lazy release consistency in the following serialized model:



This model as well as an eager release consistency model both require an announcement of a release (as the name implies) when certain conditions are met. In the eager model, that condition requires that a write would be read by all read processes in a consistent manner.

In Go, there exists alternatives for this, but there are also packages out there if you're interested in playing with it.

Using memcached

If you're not familiar with memcache(d), it's a wonderful and seemingly obvious way to manage data across distributed systems. Go's built-in channels and goroutines are fantastic to manage communication and data integrity within a single machine's processes, but neither are built for distributed systems out of the box.

Memcached, as the name implies, allows data sharing memory among multiple instances or machines. Initially, memcached was intended to store data for quick retrieval. This is useful for caching data for systems with high turnover such as web applications, but it's also a great way to easily share data across multiple servers and/or to utilize shared locking mechanisms.

In our earlier models, memcached falls under DSM. All available and invoked instances share a common, mirrored memory space within their respective memories.

It's worth pointing out that race conditions can and do exist within memcached, and you still need a way to deal with that. Memcached provides one method to share data across distributed systems, but does not guarantee data atomicity. Instead, memcached operates on one of two methods for invalidating cached data as follows:

- Data is explicitly assigned a maximum age (after which, it is removed from the stack)
- Or data is pushed from the stack due to all available memory being used by newer data

It's important to note that storage within memcache(d) is, obviously, ephemeral and not fault resistant, so it should only be used where data should be passed without threat of critical application failure.

At the point where either of these conditions is met, the data disappears and the next call to this data will fail, meaning the data needs to be regenerated. Of course, you can work with some elaborate lock generation methods to make memcached operate in a consistent manner, although this is not standard built-in functionality of memcached itself. Let's look at a quick example of memcached in Go using Brad Fitz's gomemcache interface (<https://github.com/bradfitz/gomemcache>):

```
package main

import (
    "github.com/bradfitz/gomemcache/memcache"
    "fmt"
)
```

```
func main() {  
    mC := memcache.New("10.0.0.1:11211", "10.0.0.2:11211",  
        "10.0.0.3:11211", "10.0.0.4:11211")  
    mC.Set(&memcache.Item{Key: "data", Value: []byte("30") })  
  
    dataItem, err := mc.Get("data")  
}
```

As you might note from the preceding example, if any of these memcached clients are writing to the shared memory at the same time, a race condition could still exist.

The key data can exist across any of the clients that have memcached connected and running at the same time.

Any client can also unset or overwrite the data at any time.

Unlike a lot of implementations, you can set some more complex types through memcached, such as structs, assuming they are serialized. This caveat means that we're somewhat limited with the data we can share directly. We are obviously unable to use pointers as memory locations will vary from client to client.

One method to handle data consistency is to design a master-slave system wherein only one node is responsible for writes and the other clients listen for changes via a key's existence.

We can utilize any other earlier mentioned models to strictly manage a lock on this data, although it can get especially complicated. In the next chapter, we'll explore some ways by which we can build distributed mutual exclusion systems, but for now, we'll briefly look at an alternative option.

Circuit

An interesting third-party library to handle distributed concurrency that has popped up recently is Petar Maymounkov's Go' circuit. Go' circuit attempts to facilitate distributed coroutines by assigning channels to listen to one or more remote goroutines.

The coolest part of Go' circuit is that simply including the package makes your application ready to listen and operate on remote goroutines and work with channels with which they are associated.

Go' circuit is in use at Tumblr, which proves it has some viability as a large-scale and relatively mature solutions platform.



Go' circuit can be found at <https://github.com/gocircuit/circuit>.

Installing Go' circuit is not simple—you cannot run a simple `go get` on it—and requires Apache Zookeeper and building the toolkit from scratch.

Once done, it's relatively simple to have two machines (or two processes if running locally) running Go code to share a channel. Each cog in this system falls under a sender or listener category, just as with goroutines. Given that we're talking about network resources here, the syntax is familiar with some minor modifications:

```
homeChannel := make(chan bool)

circuit.Spawn("etphonehome.example.com", func() {
    homeChannel <- true
})

for {
    select {
        case response := <- homeChannel:
            fmt.Print("E.T. has phoned home with:", response)

    }
}
```

You can see how this might make the communication between disparate machines playing with the same data a lot cleaner, whereas we used memcached primarily as a networked in-memory locking system. We're dealing with native Go code directly here; we have the ability to use circuits like we would in channels, without worrying about introducing new data management or atomicity issues. In fact, the circuit is built upon a goroutine itself.

This does, of course, still introduce some additional management issues, primarily as it pertains to knowing what remote machines are out there, whether they are active, updating the machines' statuses, and so on. These types of issues are best suited for a suite such as Apache Zookeeper to handle coordination of distributed resources. It's worth noting that you should be able to produce some feedback from a remote machine to a host: the circuit operates via passwordless SSH.

That also means you may need to make sure that user rights are locked down and that they meet with whatever security policies you may have in place.



You can find Apache Zookeeper at <http://zookeeper.apache.org/>.



Summary

Equipped now with some methods and models to manage not only local data across single or multithreaded systems, but also distributed systems, you should start to feel pretty comfortable with protecting the validity of data in concurrent and parallel processes.

We've looked at both forms of mutual exclusions for read and read/write locks, and we have started to apply these to distributed systems to prevent blocks and race conditions across multiple networked systems.

In the next chapter, we'll explore these exclusion and data consistency concepts a little deeper, building non-blocking networked applications and learn to work with timeouts and give parallelism with channels a deeper look.

We'll also dig a little deeper into the sync and OS packages, in particular looking at the `sync.atomic` operations.

5

Locks, Blocks, and Better Channels

Now that we're starting to get a good grasp of utilizing goroutines in safe and consistent ways, it's time to look a bit more at what causes code blocking and deadlocks. Let's also explore the `sync` package and dive into some profiling and analysis.

So far, we've built some relatively basic goroutines and complementary channels, but we now need to utilize some more complex communication channels between our goroutines. To do this, we'll implement more custom data types and apply them directly to channels.

We've not yet looked at some of Go's lower-level tools for synchronization and analysis, so we'll explore `sync.atomic`, a package that—along with `sync.Mutex`—allows for more granular control over state.

Finally, we'll delve into `pprof`, a fabulous tool provided by Go that lets us analyze our binaries for detailed information about our goroutines, threads, overall heap, and blocking profiles.

Armed with some new tools and methods to test and analyze our code, we'll be ready to generate a robust, highly-scalable web server that can be used to safely and quickly handle any amount of traffic thrown at it.

Understanding blocking methods in Go

So far, we've encountered a few pieces of blocking code, intentional and unintentional, through our exploration and examples. At this point, it's prudent to look at the various ways we can introduce (or inadvertently fall victim to) blocking code.

By looking at the various ways Go code can be blocked, we can also be better prepared to debug cases when concurrency is not operating as expected in our application.

Blocking method 1 – a listening, waiting channel

The most concurrently-focused way to block your code is by leaving a serial channel listening to one or more goroutines. We've seen this a few times by now, but the basic concept is shown in the following code snippet:

```
func thinkAboutKeys() {
    for {
        fmt.Println("Still Thinking")
        time.Sleep(1 * time.Second)
    }
}

func main() {
    fmt.Println("Where did I leave my keys?")

    blockChannel := make(chan int)
    go thinkAboutKeys()

    <-blockChannel

    fmt.Println("OK I found them!")
}
```

Despite the fact that all of our looping code is concurrent, we're waiting on a signal for our `blockChannel` to continue linear execution. We can, of course, see this in action by sending along the channel, thus continuing code execution as shown in the following code snippet:

```
func thinkAboutKeys(bC chan int) {
    i := 0
    max := 10
    for {
        if i >= max {
            bC <- 1
        }
        fmt.Println("Still Thinking")
        time.Sleep(1 * time.Second)
        i++
    }
}
```

Here, we've modified our goroutine function to accept our blocking channel and deliver an end message to it when we've hit our maximum. These kinds of mechanisms are important for long-running processes because we may need to know when and how to kill them.

Sending more data types via channels

Go's use of channels (structs and functions) as first-class citizens provides us with a lot of interesting ways of executing, or at least trying, new approaches of communication between channels.

One such example is to create a channel that handles translation through a function itself, and instead of communicating directly through the standard syntax, the channel executes its function. You can even do this on a slice/array of functions iterating through them in the individual functions.

Creating a function channel

So far, we've almost exclusively worked in single data type and single value channels. So, let's try sending a function across a channel. With first-class channels, we need no abstraction to do this; we can just send almost anything directly over a channel as shown in the following code snippet:

```
func abstractListener(fxChan chan func() string ) {

    fxChan <- func() string {

        return "Sent!"
    }
}

func main() {

    fxChan := make (chan func() string)
    defer close(fxChan)
    go abstractListener(fxChan)
    select {
        case rfx := <- fxChan:
            msg := rfx()
            fmt.Println(msg)
            fmt.Println("Received!")
    }

}
```

This is like a callback function. However, it also is intrinsically different, as it is not just the method called after the execution of a function, but also serves as the mode of communication between functions.

Keep in mind that there are often alternatives to passing functions across channels, so this will likely be something very specific to a use case rather than a general practice.

Since your channel's type can be virtually any available type, this functionality opens up a world of possibilities, which can be potentially confusing abstractions. A struct or interface as a channel type is pretty self-explanatory, as you can make application-related decisions on any of its defined properties.

Let's see an example of using an interface in this way in the next section.

Using an interface channel

As with our function channel, being able to pass an interface (which is a complementary data type) across a channel can be incredibly useful. Let's look at an example of sending across an interface:

```
type Messenger interface {
    Relay() string
}

type Message struct {
    status string
}

func (m Message) Relay() string {
    return m.status
}

func alertMessages(v chan Messenger, i int) {
    m := new(Message)
    m.status = "Done with " + strconv.FormatInt(int64(i), 10)
    v <- m
}

func main () {

    msg := make(chan Messenger)

    for i:= 0; i < 10; i++ {
        go alertMessages(msg,i)
    }

    select {
        case message := <-msg:
            fmt.Println (message.Relay())
    }
    <- msg
}
```

This is a very basic example of how to utilize interfaces as channels; in the previous example, the interface itself is largely ornamental. In actuality, we're passing newly-created message types through the interface's channel rather than interacting directly with the interface.

Using structs, interfaces, and more complex channels

Creating a custom type for our channel allows us to dictate the way our intra-channel communication will work while still letting Go dictate the context switching and behind-the-scenes scheduling.

Ultimately, this is mostly a design consideration. In the previous examples, we used individual channels for specific pieces of communication in lieu of a one-size-fits-all channel that passes a multitude of data. However, you may also find it advantageous to use a single channel to handle a large amount of communication between goroutines and other channels.

The primary consideration in deciding whether to segregate channels into individual bits of communication or a package of communications depends on the aggregate mutability of each.

For example, if you'll always want to send a counter along with a function or string and they will always be paired in terms of data consistency, such a method might make sense. If any of those components can lose synchronicity en route, it's more logical to keep each piece independent.

Maps in Go

As mentioned, maps in Go are like hash tables elsewhere and immediately related to slices or arrays.

In the previous example we were checking to see if a username/key exists already; for this purpose Go provides a simple method for doing so. When attempting to retrieve a hash with a nonexistent key, a zero value is returned, as shown in the following lines of code:



```
if Users[user.name] {  
    fmt.Fprintln(conn, "Unfortunately, that username  
is in  
    use!");  
}
```

This makes it syntactically simple and clean to test against a map and its keys.

One of the best features of maps in Go is the ability to make keys out of any comparable type, which includes strings, integers, Booleans as well as any map, struct, slice, or channel that is comprised exclusively of those types.

This one-to-many channel can work as a master-slave or broadcaster-subscriber model. We'll have a channel that listens for messages and routes them to appropriate users and a channel that listens for broadcast messages and queues them to all users.

To best demonstrate this, we'll create a simple multiuser chat system that allows Twitter style @user communication with a single user, with the ability to broadcast standard messages to all users and creates a universal broadcast chat note that can be read by all users. Both will be simple, custom type struct channels, so we can delineate various communication pieces.

Structs in Go

As a first-class, anonymous, and extensible type, a struct is one of the most versatile and useful data constructs available. It's simple to create analogs to other data structures such as databases and data stores, and while we hesitate to call them objects they can certainly be viewed as such.

The rule of thumb as it pertains to using structs within functions is to pass by reference rather than by value if the struct is particularly complex. Two points of clarification are as follows:



- Reference is in quotations because (and this is validated by Go's FAQ) technically everything in Go is passed by value. By that we mean that though a reference to a pointer still exists, at some step in the process the value(s) is copied.
- "Particularly complex" is, understandably, tough to quantify, so personal judgment might come into play. However, we can consider a simple struct one with no more than five methods or properties.

You can think of this in terms of a help desk system, and while in the present day we'd be unlikely to create a command-line interface for such a thing, eschewing the web portion allows us to gloss over all of the client-side code that isn't necessarily relevant to Go.

You could certainly take such an example and extrapolate it to the Web utilizing some frontend libraries for asynchronous functionality (such as `backbone.js` or `socket.io`).

To accomplish this, we'll need to create both a client and a server application, and we'll try to keep each as bare bone as possible. You can clearly and simply augment this to include any functionality you see fit such as making Git comments and updating a website.

We'll start with the server, which will be the most complicated part. The client application will mostly receive messages back through the socket, so much of the reading and routing logic will be invisible to the client-side of the process.

The net package – a chat server with interfaced channels

Here, we'll need to introduce a relevant package that will be required to handle most of the communication for our application(s). We've touched on the `net` package a bit while dabbling in the SVG output generation example to show concurrency—`net/http` is just a small part of a broader, more complex, and more feature-full package.

The basic components that we'll be using will be a TCP listener (server) and a TCP dialer (client). Let's look at the basic setup for these.

Server

Listening on a TCP port couldn't be easier. Simply initiate the `net.Listen()` method and handle the error as shown in the following lines of code:

```
listener, err := net.Listen("tcp", ":9000")
if err != nil {
    fmt.Println("Could not start server!")
}
```

If you get an error starting the server, check your firewall or modify the port—it's possible that something is utilizing port 9000 on your system.

As easy as that is, it's just as simple on our client/dialer side.

Client

In this case, we have everything running on localhost as shown in the following lines of code. However, in a real-world application we'd probably have an intranet address used here:

```
conn, err := net.Dial("tcp", "127.0.0.1:9000")
if err != nil {
    fmt.Println("Could not connect to server!")
}
```

In this application, we demonstrate two different ways to handle byte buffers of unknown lengths on `Read()`. The first is a rather crude method of trimming a string using `strings.TrimRight()`. This method allows you to define characters you aren't interested in counting as part of the input as shown in the following line of code. Mostly, it's whitespace characters that we can assume are unused parts of the buffer length.

```
sendMessage := []byte(cM.name + ": " +
    strings.TrimRight(string(buf), "\t\r\n"))
```

Dealing with strings this way is often both inelegant and unreliable. What happens if we get something we don't expect here? The string will be the length of the buffer, which in this case is 140 bytes.

The other way we deal with this is by using the end of the buffer directly. In this case, we assign the `n` variable to the `conn.Read()` function, and then can use that as a buffer length in the string to buffer conversion as shown in the following lines of code:

```
messBuff := make([]byte, 1024)
n, err := conn.Read(messBuff)
if err != nil {
}
message := string(messBuff[:n])
```

Here we're taking the first `n` bytes of the message buffer's received value.

This is more reliable and efficient, but you will certainly run into text ingestion cases where you will want to remove certain characters to create cleaner input.

Each connection in this application is a struct and each user is as well. We keep track of our users by pushing them to the `Users` slice as they join.

The selected username is a command-line argument as follows:

```
./chat-client nathan
chat-client.exe nathan
```

We do not check to ensure there is only one user with that name, so that logic might be required, particularly if chats with direct messages contain sensitive information.

Handling direct messages

For the most part, this chat client is a simple echo server, but as mentioned, we also include an ability to do non-globally broadcast messages by invoking the Twitter style `@` syntax.

We handle this mainly through regular expressions, wherein if a message matches `@user` then only that user will see the message; otherwise, it's broadcasted to all. This is somewhat inelegant, because senders of the direct message will not see their own direct message if their usernames do not match the intended names of the users.

To do this, we direct every message through a `evalMessageRecipient()` function before broadcasting. As this is relying on user input to create the regular expression (in the form of the username), please take note that we should escape this with the `regexp.QuoteMeta()` method to prevent regex failures.

Let's first examine our chat server, which is responsible for maintaining all connections and passing them to goroutines to listen and receive, as shown in the following code:

```
chat-server.go
package main

import (
    "fmt"
    "strings"
    "net"
    "strconv"
    "regexp"
)

var connectionCount int
var messagePool chan(string)

const (
    INPUT_BUFFER_LENGTH = 140
)
```

We utilize a maximum character buffer. This restricts our chat messages to no more than 140 characters. Let's look at our `User` struct to see the information we might keep about a user that joins, as follows:

```
type User struct {
    Name string
    ID int
    Initiated bool
}
```

The initiated variable tells us that `User` is connected after a connection and announcement. Let's examine the following code to understand the way we'd listen on a channel for a logged-in user:

```

    UChannel chan []byte
    Connection *net.Conn
}

```

The `User` struct contains all of the information we will maintain for each connection. Keep in mind here we don't do any sanity checking to make sure a user doesn't exist - this doesn't necessarily pose a problem in an example, but a real chat client would benefit from a response should a user name already be in use.

```

func (u *User) Listen() {
    fmt.Println("Listening for",u.Name)
    for {
        select {
            case msg := <- u.UChannel:
                fmt.Println("Sending new message to",u.Name)
                fmt.Fprintln(*u.Connection,string(msg))
        }
    }
}

```

This is the core of our server: each `User` gets its own `Listen()` method, which maintains the `User` struct's channel and sends and receives messages across it. Put simply, each user gets a concurrent channel of his or her own. Let's take a look at the `ConnectionManager` struct and the `Initiate()` function that creates our server in the following code:

```

type ConnectionManager struct {
    name      string
    initiated bool
}

func Initiate() *ConnectionManager {
    cM := &ConnectionManager{
        name:      "Chat Server 1.0",
        initiated: false,
    }

    return cM
}

```

Our `ConnectionManager` struct is initiated just once. This sets some relatively ornamental attributes, some of which could be returned on request or on chat login. We'll examine the `evalMessageRecipient` function that attempts to roughly identify the intended recipient of any message sent as follows:

```
func evalMessageRecipient(msg []byte, uName string) bool {
    eval := true
    expression := "@"
    re, err := regexp.MatchString(expression, string(msg))
    if err != nil {
        fmt.Println("Error:", err)
    }
    if re == true {
        eval = false
        pmExpression := "@" + uName
        pmRe, pmErr := regexp.MatchString(pmExpression, string(msg))
        if pmErr != nil {
            fmt.Println("Regex error", err)
        }
        if pmRe == true {
            eval = true
        }
    }
    return eval
}
```

This is our router of sorts taking the @ part of the string and using it to detect an intended recipient to hide from public consumption. We do not return an error if the user doesn't exist or has left the chat.



The format for regular expressions using the `regexp` package relies on the `re2` syntax, which is described at <https://code.google.com/p/re2/wiki/Syntax>.

Let's take a look at the code for the `Listen()` method of the `ConnectionManager` struct:

```
func (cM *ConnectionManager) Listen(listener net.Listener) {
    fmt.Println(cM.name, "Started")
    for {

        conn, err := listener.Accept()
        if err != nil {
            fmt.Println("Connection error", err)
        }
    }
}
```

```

        connectionCount++
        fmt.Println(conn.RemoteAddr(), "connected")
        user := User{Name: "anonymous", ID: 0, Initiated: false}
        Users = append(Users, &user)
        for _, u := range Users {
            fmt.Println("User online", u.Name)
        }
        fmt.Println(connectionCount, "connections active")
        go cM.messageReady(conn, &user)
    }
}

func (cM *ConnectionManager) messageReady(conn net.Conn, user
    *User) {
    uChan := make(chan []byte)

    for {

        buf := make([]byte, INPUT_BUFFER_LENGTH)
        n, err := conn.Read(buf)
        if err != nil {
            conn.Close()
            conn = nil
        }
        if n == 0 {
            conn.Close()
            conn = nil
        }
        fmt.Println(n, "character message from user", user.Name)
        if user.Initiated == false {
            fmt.Println("New User is", string(buf))
            user.Initiated = true
            user.UChannel = uChan
            user.Name = string(buf[:n])
            user.Connection = &conn
            go user.Listen()

            minusYouCount := strconv.FormatInt(int64(connectionCount-1),
                10)
            conn.Write([]byte("Welcome to the chat, " + user.Name + ",
                there are " + minusYouCount + " other users"))

        } else {

```

```
        sendMessage := []byte(user.Name + ": " +
            strings.TrimRight(string(buf), " \t\r\n"))

        for _, u := range Users {
            if evalMessageRecipient(sendMessage, u.Name) == true {
                u.UChannel <- sendMessage
            }
        }
    }

    }
}geReady (per connectionManager) function instantiates new
connections into a User struct, utilizing first sent message as
the user's name.

var Users []*User
This is our unbuffered array (or slice) of user structs.
func main() {
    connectionCount = 0
    serverClosed := make(chan bool)

    listener, err := net.Listen("tcp", ":9000")
    if err != nil {
        fmt.Println ("Could not start server!",err)
    }

    connManage := Initiate()
    go connManage.Listen(listener)

    <-serverClosed
}
```

As expected, `main()` primarily handles the connection and error and keeps our server open and nonblocked with the `serverClosed` channel.

There are a number of methods we could employ to improve the way we route messages. The first method would be to invoke a map (or hash table) bound to a username. If the map's key exists, we could return some error functionality if a user already exists, as shown in the following code snippet:

```
type User struct {
    name string
}
var Users map[string] *User

func main() {
    Users := make(map[string] *User)
}
```

Examining our client

Our client application is a bit simpler primarily because we don't care as much about blocking code.

While we do have two concurrent operations (wait for the message and wait for user input to send the message), this is significantly less complicated than our server, which needs to concurrently listen to each created user and distribute sent messages, respectively.

Let's now compare our chat client to our chat server. Obviously, the client has less overall maintenance of connections and users, and so we do not need to use nearly as many channels. Let's take a look at our chat client's code:

```
chat-client.go
package main

import (
    "fmt"
    "net"
    "os"
    "bufio"
    "strings"
)
```



```
type Message struct {
    message string
    user string
}

var recvBuffer [140]byte

func listen(conn net.Conn) {
    for {

        messBuff := make([]byte, 1024)
        n, err := conn.Read(messBuff)
        if err != nil {
            fmt.Println("Read error", err)
        }
        message := string(messBuff[:n])
        message = message[0:]

        fmt.Println(strings.TrimSpace(message))
        fmt.Print("> ")
    }
}

func talk(conn net.Conn, mS chan Message) {

    for {
        command := bufio.NewReader(os.Stdin)
        fmt.Print("> ")
        line, err := command.ReadString('\n')

        line = strings.TrimRight(line, " \t\r\n")
        _, err = conn.Write([]byte(line))
        if err != nil {
            conn.Close()
            break
        }
        doNothing(command)
    }
}
```

```
func doNothing(bf *bufio.Reader) {
    // A temporary placeholder to address io reader usage

}

func main() {

    messageServer := make(chan Message)

    userName := os.Args[1]

    fmt.Println("Connecting to host as",userName)

    clientClosed := make(chan bool)

    conn, err := net.Dial("tcp","127.0.0.1:9000")
    if err != nil {
        fmt.Println("Could not connect to server!")
    }
    conn.Write([]byte(userName))
    introBuff := make([]byte,1024)
    n, err := conn.Read(introBuff)
    if err != nil {

    }
    message := string(introBuff[:n])
    fmt.Println(message)

    go talk(conn,messageServer)
    go listen(conn)

    <- clientClosed
}
```

Blocking method 2 – the select statement in a loop

Have you noticed yet that the `select` statement itself blocks? Fundamentally, the `select` statement is not different from an open listening channel; it's just wrapped in conditional code.

The `<- myChannel` channel operates the same way as the following code snippet:

```
select {
    case mc := <- myChannel:
        // do something
}
```

An open listening channel is not a deadlock as long as there are no goroutines sleeping. You'll find this on channels that are listening but will never receive anything, which is another method of basically waiting.

These are useful shortcuts for long-running applications you wish to keep alive but you may not necessarily need to send anything along that channel.

Cleaning up goroutines

Any channel that is left waiting and/or left receiving will result in a deadlock. Luckily, Go is pretty adept at recognizing these and you will almost without fail end up in a panic when running or building the application.

Many of our examples so far have utilized the deferred `close()` method of immediately and cleanly grouping together similar pieces of code that should execute at different points.

While garbage collection handles a lot of the cleanup, we're largely left to take care of open channels to ensure we don't have a process waiting to receive and/or something waiting to send, both waiting at the same time for each other. Luckily, we'll be unable to compile any such program with a detectable deadlock condition, but we also need to manage closing channels that are left waiting.

Quite a few of the examples so far have ended with a generic integer or Boolean channel that just waits—this is employed almost exclusively for the channel's blocking effect and allows us to demonstrate the effects and output of concurrent code while the application is still running. In many cases, this generic channel is an unnecessary bit of syntactical cruft as shown in the following lines of code:

```
<-youMayNotNeedToDoThis
close(youmayNotNeedToDoThis)
```

The fact that there's no assignment happening is a good indicator this is an example of such cruft. If we had instead modified that to include an assignment, the previous code would be changed to the following instead:

```
v := <-youMayNotNeedToDoThis
```

It might indicate that the value is useful and not just arbitrary blocking code.

Blocking method 3 – network connections and reads

If you run the code from our earlier chat server's client without starting the server, you'll notice that the `Dial` function blocks any subsequent goroutine. We can test this by imposing a longer-than-normal timeout on the connection or by simply closing the client application after logging in, as we did not implement a method for closing the TCP connection.

As the network reader we're using for the connection is buffered, we'll always have a blocking mechanism while waiting for data via TCP.

Creating channels of channels

The preferred and sanctioned way of managing concurrency and state is exclusively through channels.

We've demonstrated a few more complex types of channels, but we haven't looked at what can become a daunting but powerful implementation: channels of channels. This might at first sound like some unmanageable wormhole, but in some situations we want a concurrent action to generate more concurrent actions; thus, our goroutines should be capable of spawning their own.

As always, the way you manage this is through design while the actual code may simply be an aesthetic byproduct here. Building an application this way should make your code more concise and clean most of the time.

Let's revisit a previous example of an RSS feed reader to demonstrate how we could manage this, as shown in the following code:

```
package main

import (
    "fmt"
)

type master chan Item

var feedChannel chan master
var done chan bool
```

```
type Item struct {
    Url string
    Data []byte
}

type Feed struct {
    Url string
    Name string
    Items []Item
}

var Feeds []Feed

func process(feedChannel *chan master, done *chan bool) {
    for _, i := range Feeds {
        fmt.Println("feed", i)
        item := Item{}
        item.Url = i.Url
        itemChannel := make(chan Item)
        *feedChannel <- itemChannel
        itemChannel <- item
    }
    *done <- true
}

func processItem(url string) {
    // deal with individual feed items here
    fmt.Println("Got url", url)
}

func main() {
    done := make(chan bool)
    Feeds = []Feed{Feed{Name: "New York Times", Url: "http://rss.nytimes.
com/services/xml/rss/nyt/HomePage.xml"},
        Feed{Name: "Wall Street Journal", Url: "http://feeds.wsjonline.com/
wsj/xml/rss/3_7011.xml"}}
    feedChannel := make(chan master)
    go func(done chan bool, feedChannel chan master) {
        for {
            select {
            case fc := <-feedChannel:
                select {
                case item := <-fc:
                    processItem(item.Url)
                }
            }
        }
    }(done, feedChannel)
```

```
        default:
        }
    }
} (done, feedChannel)
go process(&feedChannel, &done)
<-done
fmt.Println("Done!")
}
```

Here, we manage `feedChannel` as a custom struct that is itself a channel for our `Item` type. This allows us to rely exclusively on channels for synchronization handled through a semaphore-esque construct.

If we want to look at another way of handling a lower-level synchronization, `sync.atomic` provides some simple iterative patterns that allow you to manage synchronization directly in memory.

As per Go's documentation, these operations require great care and are prone to data consistency errors, but if you need to touch memory directly, this is the way to do it. When we talk about advanced concurrency features, we'll utilize this package directly.

Pprof – yet another awesome tool

Just when you think you've seen the entire spectrum of Go's amazing tool set, there's always one more utility that, once you realize it exists, you'll wonder how you ever survived without it.

Go format is great for cleaning up your code; the `-race` flag is essential for detecting possible race conditions, but an even more robust, hands-in-the-dirt tool exists that is used to analyze your final application, and that is pprof.

Google created pprof initially to analyze loop structures and memory allocation (and related types) for C++ applications.

It's particularly useful if you think you have performance issues not uncovered by the testing tools provided in the Go runtime. It's also a fantastic way to generate a visual representation of the data structures in any application.

Some of this functionality also exists as part of the Go testing package and its benchmarking tools – we'll explore that more in *Chapter 7, Performance and Scalability*.

Getting the runtime version of pprof to work requires a few pieces of setup first. We'll need to include the `runtime.pprof` package and the `flag` package, which allows command-line parsing (in this case, for the output of pprof).

If we take our chat server code, we can add a couple of lines and have the application prepped for performance profiling.

Let's make sure we include those two packages along with our other packages. We can use the underscore syntax to indicate to the compiler that we're only interested in the package's side effects (meaning we get the package's initialization functions and global variables) as shown in the following lines of code:

```
import
(
    "fmt"
    ...
    _ "runtime/pprof"
)
```

Next, in our `main()` function, we include a flag parser that will parse and interpret the data produced by `pprof` as well as create the CPU profile itself if it does not exist (and bailing if it cannot be created), as shown in the following code snippet:

```
var profile = flag.String("cpuprofile", "", "output pprof data to
file")

func main() {
    flag.Parse()
    if *profile != "" {
        flag, err := os.Create(*profile)
        if err != nil {
            fmt.Println("Could not create profile", err)
        }
        pprof.StartCPUProfile(flag)
        defer pprof.StopCPUProfile()
    }
}
```

This tells our application to generate a CPU profiler if it does not exist, start the profiling at the beginning of the execution, and defer the end of the profiling until the application exits successfully.

With this created, we can run our binary with the `cpuprofile` flag, which tells the program to generate a profile file as follows:

```
./chat-server -cpuprofile=chat.prof
```

For the sake of variety (and exploiting more resources arbitrarily), we'll abandon the chat server for a moment and create a loop generating scores of goroutines before exiting. This should give us a more exciting demonstration of profiling data than a simple and long-living chat server would, although we'll return to that briefly:

Here is our example code that generates more detailed and interesting profiling data:

```
package main

import (
    "flag"
    "fmt"
    "math/rand"
    "os"
    "runtime"
    "runtime/pprof"
)

const ITERATIONS = 99999
const STRINGLENGTH = 300

var profile = flag.String("cpuprofile", "", "output pprof data to
file")

func generateString(length int, seed *rand.Rand, chHater chan
string) string {
    bytes := make([]byte, length)
    for i := 0; i < length; i++ {
        bytes[i] = byte(rand.Int())
    }
    chHater <- string(bytes[:length])
    return string(bytes[:length])
}

func generateChannel() <-chan int {
    ch := make(chan int)
    return ch
}

func main() {

    goodbye := make(chan bool, ITERATIONS)
    channelThatHatesLetters := make(chan string)

    runtime.GOMAXPROCS(2)
    flag.Parse()
    if *profile != "" {
        flag, err := os.Create(*profile)
        if err != nil {
            fmt.Println("Could not create profile", err)
        }
        pprof.StartCPUProfile(flag)
        defer pprof.StopCPUProfile()
    }
}
```



```
seed := rand.New(rand.NewSource(19))

initString := ""

for i := 0; i < ITERATIONS; i++ {
    go func() {
        initString = generateString(STRINGLENGTH, seed,
            channelThatHatesLetters)
        goodbye <- true
    }()

}
select {
case <-channelThatHatesLetters:

}
<-goodbye

fmt.Println(initString)
}
```

When we generate a profile file out of this, we can run the following command:

```
go tool pprof chat-server chat-server.prof
```

This will start the pprof application itself. This gives us a few commands that report on the static, generated file as follows:

- `topN`: This shows the top *N* samples from the profile file, where *N* represents the explicit number you want to see.
- `web`: This creates a visualization of data, exports it to SVG, and opens it in a web browser. To get the SVG output, you'll need to install Graphviz as well (<http://www.graphviz.org/>).



You can also run pprof with some flags directly to output in several formats or launch a browser as follows:

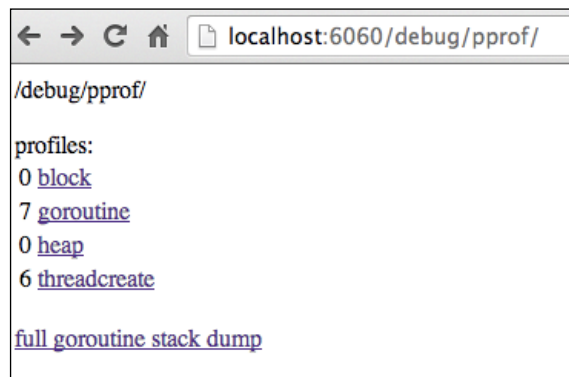
- `--text`: This generates a text report
- `--web`: This generates an SVG and opens in the browser
- `--gv`: This generates the Ghostview postscript
- `--pdf`: This generates the PDF to output
- `--SVG`: This generates the SVG to output
- `--gif`: This generates the GIF to output



And voila, we suddenly have an insight into how our program utilizes the CPU time consumption and a general view of how our application executes, loops, and exits.

In typical Go fashion, the pprof tool also exists in the `net/http` package, although it's more data-centric than visual. This means that rather than dealing exclusively with a command-line tool, you can output the results directly to the Web for analysis.

Like the command-line tool, you'll see block, goroutine, heap, and thread profiles as well as a full stack outline directly through localhost, as shown in the following screenshot:



To generate this server, you just need to include a few key lines of code in your application, build it, and then run it. For this example, we've included the code in our chat server application, which allows us to get the Web view of an otherwise command-line-only application.

Make sure you have the `net/http` and `log` packages included. You'll also need the `http/pprof` package. The code snippet is as follows:

```
import(_(
    "net/http/pprof"
    "log"
    "net/http"
)
```

Then simply include this code somewhere in your application, ideally, near the top of the `main()` function, as follows:

```
go func() {
    log.Println(http.ListenAndServe("localhost:6060", nil))
}()
```

As always, the port is entirely a matter of preference.

You can then find a number of profiling tools at `localhost:6060`, including the following:

- All tools can be found at `http://localhost:6060/debug/pprof/`
- Blocking profiles can be found at `http://localhost:6060/debug/pprof/block?debug=1`
- A profile of all goroutines can be found at `http://localhost:6060/debug/pprof/goroutine?debug=1`
- A detailed profile of the heap can be found at `http://localhost:6060/debug/pprof/heap?debug=1`
- A profile of threads created can be found at `http://localhost:6060/debug/pprof/threadcreate?debug=1`

In addition to the blocking profile, you may find a utility to track down inefficiency in your concurrent strategy through the thread creation profile. If you find a seemingly abnormal amount of threads created, you can toy with the synchronization structure as well as runtime parameters to streamline this.

Keep in mind that using pprof this way will also include some analyses and profiles that can be attributed to the `http` or `pprof` packages rather than your core code. You will find certain lines that are quite obviously not part of your application; for example, a thread creation analysis of our chat server includes a few telling lines, as follows:

```
#      0x7765e      net/http.HandlerFunc.ServeHTTP+0x3e      /usr/
local/go/src/pkg/net/http/server.go:1149
#      0x7896d      net/http.(*ServeMux).ServeHTTP+0x11d /usr/
local/go/src/pkg/net/http/server.go:1416
```

Given that we specifically eschewed delivering our chat application via HTTP or web sockets in this iteration, this should be fairly evident.

On top of that, there are even more obvious smoking guns, as follows:

```
#      0x139541      runtime/pprof.writeHeap+0x731      /usr/
local/go/src/pkg/runtime/pprof/pprof.go:447
#      0x137aa2      runtime/pprof.(*Profile).WriteTo+0xb2 /usr/
local/go/src/pkg/runtime/pprof/pprof.go:229
#      0x9f55f      net/http/pprof.handler.ServeHTTP+0x23f /usr/
local/go/src/pkg/net/http/pprof/pprof.go:165
#      0x9f6a5      net/http/pprof.Index+0x135      /usr/
local/go/src/pkg/net/http/pprof/pprof.go:177
```

Some system and Go core mechanisms we will never be able to reduce out of our final compiled binaries are as follows:

```
#      0x18d96 runtime.starttheworld+0x126
      /usr/local/go/src/pkg/runtime/proc.c:451
```



The hexadecimal value represents the address in the memory of the function when run.



A note for Windows users: pprof is a breeze to use in *nix environments but may take some more arduous tweaking under Windows. Specifically, you may need a bash replacement such as Cygwin. You may also find some necessary tweaks to pprof itself (in actuality, a Perl script) may be in order. For 64-bit Windows users, make sure you install ActivePerl and execute the pprof Perl script directly using the 64-bit version of Perl.

At publish time, there are also some issues running this on 64-bit OSX.

Handling deadlocks and errors

Anytime you encounter a deadlock error upon compilation in your code, you'll see the familiar string of semi-cryptic errors explaining which goroutine was left holding the bag, so to speak.

However, keep in mind you always have the ability to invoke your own panic using Go's built-in panic, and this can be incredibly useful for building your own error-catching safeguards to ensure data consistency and ideal operation. The code is as follows:

```
package main

import
(
    "os"
)

func main() {
    panic("Oh No, we forgot to write a program!")
    os.Exit(1)
}
```

This can be utilized anywhere you wish to give detailed exit information to either developers or end users.

Summary

Having explored some new ways to examine the way that Go code can block and deadlock, we also have some tools at our disposal that can be used to examine CPU profiles and resource usage now.

Hopefully, by this point, you can build some complex concurrent systems with simple goroutines and channels all the way up to multiplexed channels of structs, interfaces, and other channels.

We've built some somewhat-functional applications so far, but next we're going to utilize everything we've done to build a usable web server that solves a classic problem and can be used to design intranets, file storage systems, and more.

In the next chapter, we'll take what we've done in this chapter with regard to extensible channels and apply it to solving one of the oldest challenges the Internet has to offer: concurrently serving 10,000 (or more) connections.

6

C10K – A Non-blocking Web Server in Go

Up to this point, we've built a few usable applications; things we can start with and leapfrog into real systems for everyday use. By doing so, we've been able to demonstrate the basic and intermediate-level patterns involved in Go's concurrent syntax and methodology.

However, it's about time we take on a real-world problem—one that has vexed developers (and their managers and VPs) for a great deal of the early history of the Web.

In addressing and, hopefully, solving this problem, we'll be able to develop a high-performance web server that can handle a very large volume of live, active traffic.

For many years, the solution to this problem was solely to throw hardware or intrusive caching systems at the problem; so, alternately, solving it with programming methodology should excite any programmer.

We'll be using every technique and language construct we've learned so far, but we'll do so in a more structured and deliberate way than we have up to now. Everything we've explored so far will come into play, including the following points:

- Creating a visual representation of our concurrent application
- Utilizing goroutines to handle requests in a way that will scale
- Building robust channels to manage communication between goroutines and the loop that will manage them
- Profiling and benchmarking tools (JMeter, ab) to examine the way our event loop actually works
- Timeouts and concurrency controls—when necessary—to ensure data and request consistency

Attacking the C10K problem

The genesis of the C10K problem is rooted in serial, blocking programming, which makes it ideal to demonstrate the strength of concurrent programming, especially in Go.

The proposed problem came from developer Dan Kegel, who famously asked:

It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the web is a big place now.

- Dan Kegel (<http://www.kegel.com/c10k.html>)

When he asked this in 1999, for many server admins and engineers, serving 10,000 concurrent visitors was something that would be solved with hardware. The notion that a single server on common hardware could handle this type of CPU and network bandwidth without falling over seemed foreign to most.

The crux of his proposed solutions relied on producing non-blocking code. Of course, in 1999, concurrency patterns and libraries were not widespread. C++ had some polling and queuing options available via some third-party libraries and the earliest predecessor to multithreaded syntaxes, later available through Boost and then C++11.

Over the coming years, solutions to the problem began pouring in across various flavors of languages, programming design, and general approaches. At the time of publishing this book, the C10K problem is not one without solutions, but it is still an excellent platform to conduct a very real-world challenge to high-performance Go.

Any performance and scalability problem will ultimately be bound to the underlying hardware, so as always, your mileage may vary. Squeezing 10,000 concurrent connections on a 486 processor with 500 MB of RAM will certainly be more challenging than doing so on a barebones Linux server stacked with memory and multiple cores.

It's also worth noting that a simple echo server would obviously be able to assume more cores than a functional web server that returns larger amounts of data and accepts greater complexity in requests, sessions, and so on, as we'll be dealing with here.

Failing of servers at 10,000 concurrent connections

As you may recall, when we discussed concurrent strategies back in *Chapter 3, Developing a Concurrent Strategy*, we talked a bit about Apache and its load-balancing tools.

When the Web was born and the Internet commercialized, the level of interactivity was pretty minimal. If you're a graybeard, you may recall the transition from NNTP/IRC and the like and how extraordinarily rudimentary the Web was.

To address the basic proposition of [page request] → [HTTP response], the requirements on a web server in the early 1990s were pretty lenient. Ignoring all of the error responses, header readings and settings, and other essential (but unrelated to the in → out mechanism) functions, the essence of the early servers was shockingly simple, at least compared to the modern web servers.

The first web server was developed by the father of the Web, Tim Berners-Lee.

Developed at CERN (such as WWW/HTTP itself), CERN httpd handled many of the things you would expect in a web server today – hunting through the code, you'll find a lot of notation that will remind you that the very core of the HTTP protocol is largely unchanged. Unlike most technologies, HTTP has had an extraordinarily long shelf life.



Written in C in 1990, it was unable to utilize a lot of concurrency strategies available in languages such as Erlang. Frankly, doing so was probably unnecessary – the majority of web traffic was a matter of basic file retrieval and protocol. The meat and potatoes of a web server were not dealing with traffic, but rather dealing with the rules surrounding the protocol itself.

You can still access the original CERN httpd site and download the source code for yourself from <http://www.w3.org/Daemon/>. I highly recommend that you do so as both a history lesson and a way to look at the way the earliest web server addressed some of the earliest problems.

However, the Web in 1990 and the Web when the C10K question was first posed were two very different environments.

By 1999, most sites had some level of secondary or tertiary latency provided by third-party software, CGI, databases, and so on, all of which further complicated the matter. The notion of serving 10,000 flat files concurrently is a challenge in itself, but try doing so by running them on top of a Perl script that accesses a MySQL database without any caching layer; the challenge is immediately exacerbated.

By the mid 1990s, the Apache web server had taken hold and largely controlled the market (by 2009, it had become the first server software to serve more than 100 million websites).

Apache's approach was rooted heavily in the earliest days of the Internet. At its launch, connections were initially handled first in, first out. Soon, each connection was assigned a thread from the thread pool. There are two problems with the Apache server. They are as follows:

- Blocking connections can lead to a domino effect, wherein one or more slowly resolved connections could avalanche into inaccessibility
- Apache had hard limits on the number of threads/workers you could utilize, irrespective of hardware constraints

It's easy to see the opportunity here, at least in retrospect. A concurrent server that utilizes actors (Erlang), agents (Clojure), or goroutines (Go) seems to fit the bill perfectly. Concurrency does not *solve* the C10k problem in itself, but it absolutely provides a methodology to facilitate it.

The most notable and visible example of an approach to the C10K problem today is Nginx, which was developed using concurrency patterns, widely available in C by 2002 to address – and ultimately solve – the C10k problem. Nginx, today, represents either the #2 or #3 web server in the world, depending on the source.

Using concurrency to attack C10K

There are two primary approaches to handle a large volume of concurrent requests. The first involves allocating threads per connection. This is what Apache (and a few others) do.

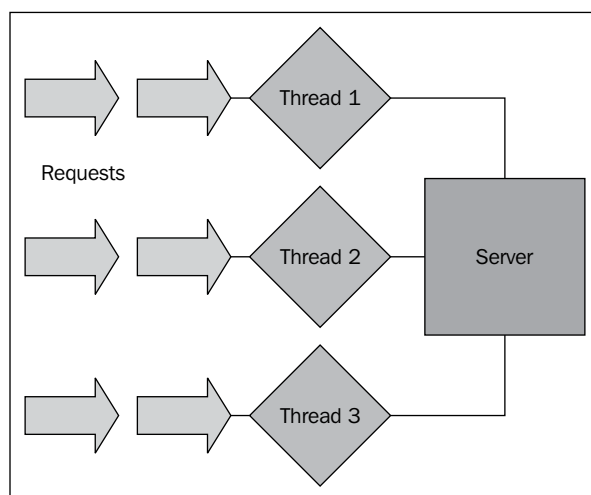
On the one hand, allocating a thread to a connection makes a lot of sense – it's isolated, controllable via the application's and kernel's context switching, and can scale with increased hardware.

One problem for Linux servers — on which the majority of the Web lives — is that each allocated thread reserves 8 MB of memory for its stack by default. This can (and should) be redefined, but this imposes a largely unattainable amount of memory required for a single server. Even if you set the default stack size to 1 MB, we're dealing with a minimum of 10 GB of memory just to handle the overhead.

This is an extreme example that's unlikely to be a real issue for a couple of reasons: first, because you can dictate the maximum amount of resources available to each thread, and second, because you can just as easily load balance across a few servers and instances rather than add 10 GB to 80 GB of RAM.

Even in a threaded server environment, we're fundamentally bound to the issue that can lead to performance decreases (to the point of a crash).

First, let's look at a server with connections bound to threads (as shown in the following diagram), and visualize how this can lead to logjams and, eventually, crashes:



This is obviously what we want to avoid. Any I/O, network, or external process that can impose some slowdown can bring about that avalanche effect we talked about, such that our available threads are taken (or backlogged) and incoming requests begin to stack up.

We can spawn more threads in this model, but as mentioned earlier, there are potential risks there too, and even this will fail to mitigate the underlying problem.

Taking another approach

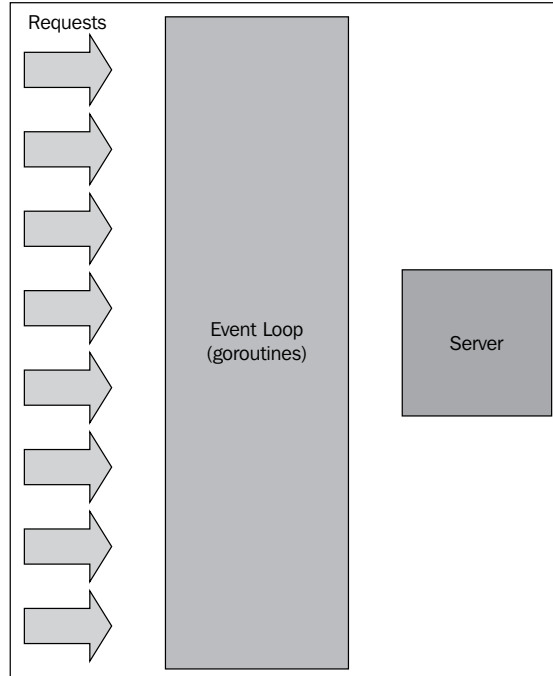
In an attempt to create our web server that can handle 10,000 concurrent connections, we'll obviously leverage our goroutine/channel mechanism to put an event loop in front of our content delivery to keep new channels recycled or created constantly.

For this example, we'll assume we're building a corporate website and infrastructure for a rapidly expanding company. To do this, we'll need to be able to serve both static and dynamic content.

The reason we want to introduce dynamic content is not just for the purposes of demonstration—we want to challenge ourselves to show 10,000 true concurrent connections even when a secondary process gets in the way.

As always, we'll attempt to map our concurrency strategy directly to goroutines and channels. In a lot of other languages and applications, this is directly analogous to an event loop, and we'll approach it as such. Within our loop, we'll manage the available goroutines, expire or reuse completed ones, and spawn new ones where necessary.

In this example visualization, we show how an event loop (and corresponding goroutines) can allow us to scale our connections without employing too many *hard* resources such as CPU threads or RAM:



The most important step for us here is to manage that event loop. We'll want to create an open, infinite loop to manage the creation and expiration of our goroutines and respective channels.

As part of this, we will also want to do some internal logging of what's happening, both for benchmarking and debugging our application.

Building our C10K web server

Our web server will be responsible for taking requests, routing them, and serving either flat files or dynamic files with templates parsed against a few different data sources.

As mentioned earlier, if we exclusively serve flat files and remove much of the processing and network latency, we'd have a much easier time with handling 10,000 concurrent connections.

Our goal is to approach as much of a real-world scenario as we can—very little of the Web operates on a single server in a static fashion. Most websites and applications utilize databases, **CDNs (Content Delivery Networks)**, dynamic and uncached template parsing, and so on. We need to replicate them whenever possible.

For the sake of simplicity, we'll separate our content by type and filter them through URL routing, as follows:

- `/static/[request]`: This will serve `request.html` directly
- `/template/[request]`: This will serve `request.tpl` after its been parsed through Go
- `/dynamic/[request][number]`: This will also serve `request.tpl` and parse it against a database source's record

By doing this, we should get a better mixture of possible HTTP request types that could impede the ability to serve large numbers of users simultaneously, especially in a blocking web server environment.

We'll utilize the `html/template` package to do parsing—we've briefly looked at the syntax before, and going any deeper is not necessarily part of the goals of this book. However, you should look into it if you're going to parlay this example into something you use in your environment or have any interest in building a framework.



You can find Go's exceptional library to generate safe data-driven templating at <http://golang.org/pkg/html/template/>.



By safe, we're largely referring to the ability to accept data and move it directly into templates without worrying about the sort of injection issues that are behind a large amount of malware and cross-site scripting.

For the database source, we'll use MySQL here, but feel free to experiment with other databases if you're more comfortable with them. Like the `html/template` package, we're not going to put a lot of time into outlining MySQL and/or its variants.

Benchmarking against a blocking web server

It's only fair to add some starting benchmarks against a blocking web server first so that we can measure the effect of concurrent versus nonconcurrent architecture.

For our starting benchmarks, we'll eschew any framework, and we'll go with our old stalwart, Apache.

For the sake of completeness here, we'll be using an Intel i5 3GHz machine with 8 GB of RAM. While we'll benchmark our final product on Ubuntu, Windows, and OS X here, we'll focus on Ubuntu for our example.

Our localhost domain will have three plain HTML files in `/static`, each trimmed to 80 KB. As we're not using a framework, we don't need to worry about raw dynamic requests, but only about static and dynamic requests in addition to data source requests.

For all examples, we'll use a MySQL database (named `master`) with a table called `articles` that will contain 10,000 duplicate entries. Our structure is as follows:

```
CREATE TABLE articles (  
    article_id INT NOT NULL AUTO_INCREMENT,  
    article_title VARCHAR(128) NOT NULL,  
    article_text VARCHAR(128) NOT NULL,  
    PRIMARY KEY (article_id)  
)
```

With ID indexes ranging sequentially from 0-10,000, we'll be able to generate random number requests, but for now, we just want to see what kind of basic response we can get out of Apache serving static pages with this machine.

For this test, we'll use Apache's `ab` tool and then `gnuplot` to sequentially map the request time as the number of concurrent requests and pages; we'll do this for our final product as well, but we'll also go through a few other benchmarking tools for it to get some better details.

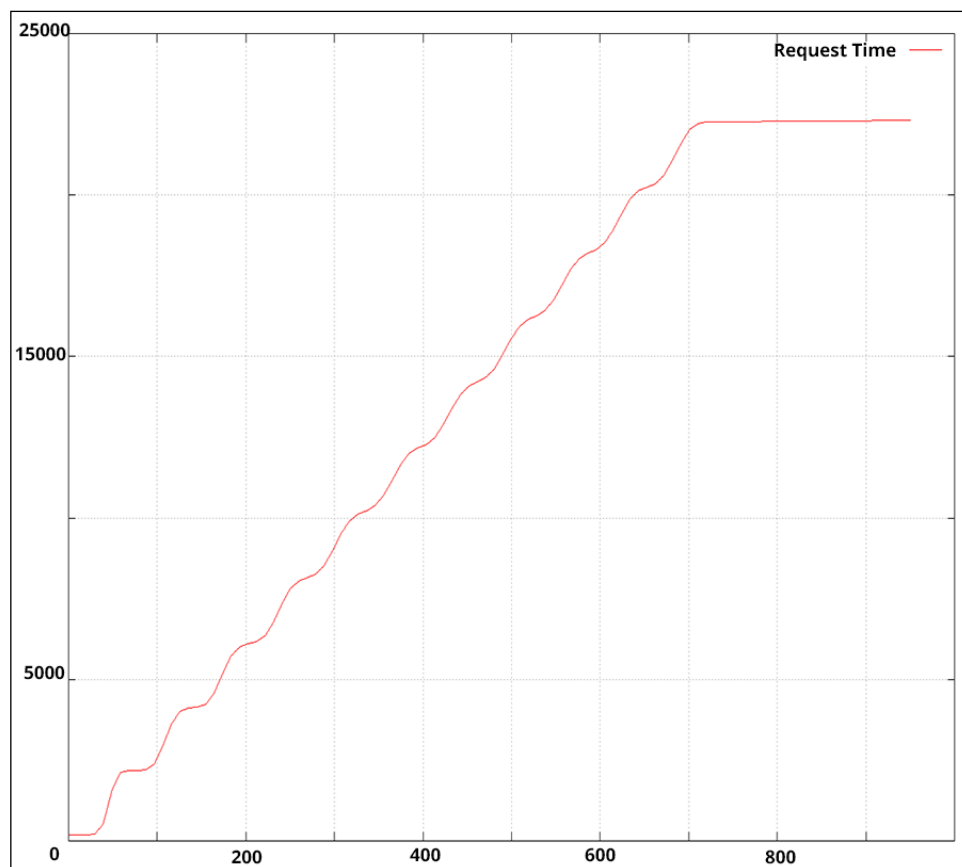


Apache's AB comes with the Apache web server itself. You can read more about it at <http://httpd.apache.org/docs/2.2/programs/ab.html>.

You can download it for Linux, Windows, OS X, and more from <http://httpd.apache.org/download.cgi>.

The gnuplot utility is available for the same operating systems at <http://www.gnuplot.info/>.

So, let's see how we did it. Have a look at the following graph:



Ouch! Not even close. There are things we can do to tune the connections available (and respective threads/workers) within Apache, but this is not really our goal. Mostly, we want to know what happens with an out-of-the-box Apache server. In these benchmarks, we start to drop or refuse connections at around 800 concurrent connections.

More troubling is that as these requests start stacking up, we see some that exceed 20 seconds or more. When this happens in a blocking server, each request behind it is queued; requests behind that are similarly queued and the entire thing starts to fall apart.

Even if we cannot hit 10,000 concurrent connections, there's a lot of room for improvement. While a single server of any capacity is no longer the way we expect a web server environment to be designed, being able to squeeze as much performance as possible out of that server, ostensibly with our concurrent, event-driven approach, should be our goal.

Handling requests

In an earlier chapter, we handled URL routing with Gorilla, a compact but feature-full framework. The Gorilla toolkit certainly makes this easier, but we should also know how to intercept the functionality to impose our own custom handler.

Here is a simple web router wherein we handle and direct requests using a custom `http.Server` struct, as shown in the following code:

```
var routes []string

type customRouter struct {
}

func (customRouter) ServeHTTP(rw http.ResponseWriter, r
    *http.Request) {

    fmt.Println(r.URL.Path);
}

func main() {

    var cr customRouter;

    server := &http.Server {
        Addr: ":9000",
        Handler:cr,
        ReadTimeout: 10 * time.Second,
        WriteTimeout: 10 * time.Second,
        MaxHeaderBytes: 1 << 20,
    }

    server.ListenAndServe()
}
```

Here, instead of using a built-in URL routing muxer and dispatcher, we're creating a custom server and custom handler type to accept URLs and route requests. This allows us to be a little more robust with our URL handling.

In this case, we created a basic, empty struct called `customRouter` and passed it to our custom server creation call.

We can add more elements to our `customRouter` type, but we really don't need to for this simple example. All we need to do is to be able to access the URLs and pass them along to a handler function. We'll have three: one for static content, one for dynamic content, and one for dynamic content from a database.

Before we go so far though, we should probably see what our absolute barebones HTTP server written in Go does when presented with the same traffic that we sent Apache's way.

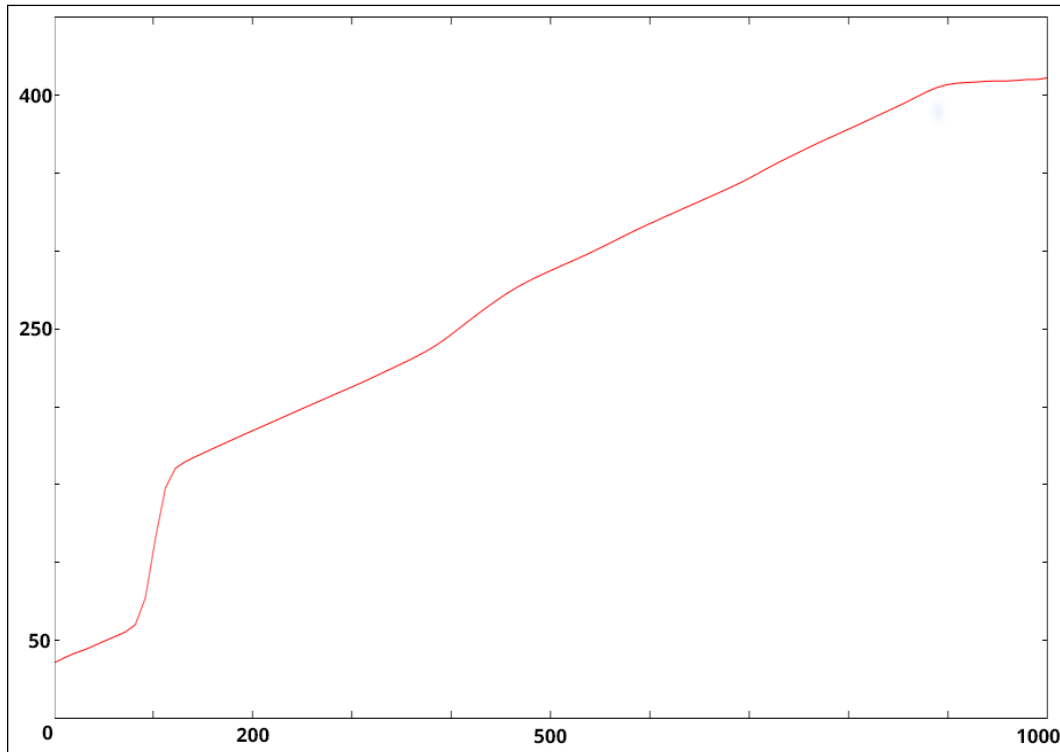
By old school, we mean that the server will simply accept a request and pass along a static, flat file. You could do this using a custom router as we did earlier, taking requests, opening files, and then serving them, but Go provides a much simpler mode to handle this basic task in the `http.FileServer` method.

So, to get some benchmarks for the most basic of Go servers against Apache, we'll utilize a simple `FileServer` and test it against a `test.html` page (which contains the same 80 KB file that we had with Apache).




As our goal with this test is to improve our performance in serving flat and dynamic pages, the actual specs for the test suite are somewhat immaterial. We'd expect that while the metrics will not match from environment to environment, we should see a similar trajectory. That said, it's only fair we supply the environment used for these tests; in this case, we used a MacBook Air with a 1.4 GHz i5 processor and 4 GB of memory.

First, we'll do this with our absolute best performance out of the box with Apache, which had 850 concurrent connections and 900 total requests.



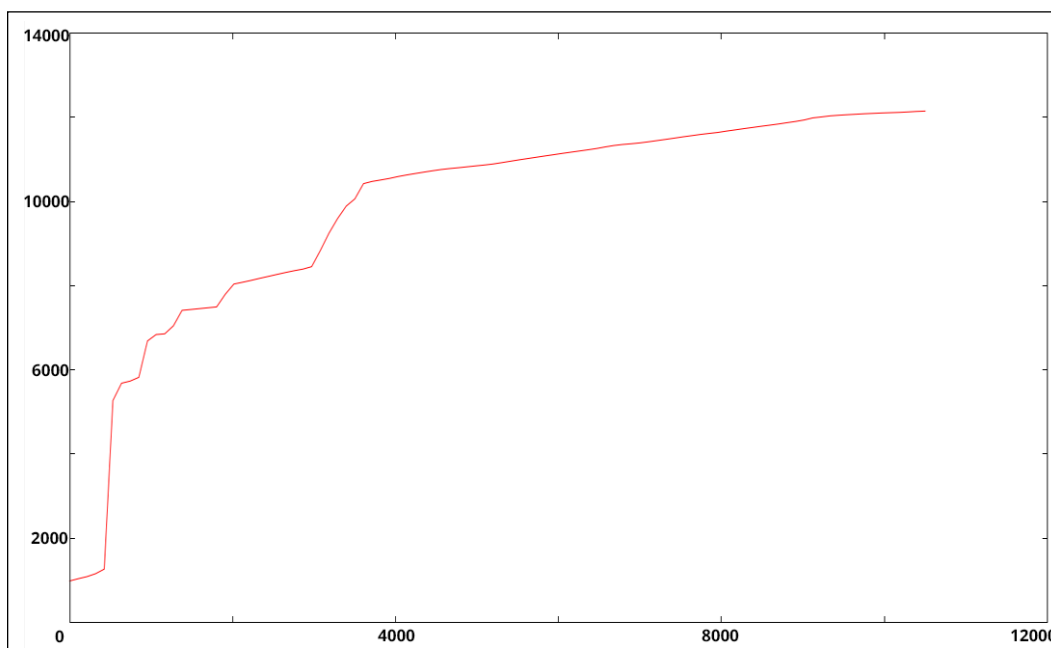
The results are certainly encouraging as compared to Apache. Neither of our test systems were tweaked much (Apache as installed and basic FileServer in Go), but Go's FileServer handles 1,000 concurrent connections without so much as a blip, with the slowest clocking in at 411 ms.

[ Apache has made a great number of strides pertaining to concurrency and performance options in the last five years, but to get there does require a bit of tuning and testing. The intent of this experiment is not intended to denigrate Apache, which is well tested and established. Instead, it's to compare the out-of-the-box performance of the world's number 1 web server against what we can do with Go.]

To really get a baseline of what we can achieve in Go, let's see if Go's FileServer can hit 10,000 connections on a single, modest machine out of the box:

```
ab -n 10500 -c 10000 -g test.csv http://localhost:8080/a.html
```

We will get the following output:



Success! Go's FileServer by itself will easily handle 10,000 concurrent connections, serving flat, static content.

Of course, this is not the goal of this particular project – we'll be implementing real-world obstacles such as template parsing and database access, but this alone should show you the kind of starting point that Go provides for anyone who needs a responsive server that can handle a large quantity of basic web traffic.

Routing requests

So, let's take a step back and look again at routing our traffic through a traditional web server to include not only our static content, but also the dynamic content.

We'll want to create three functions that will route traffic from our `customRouter`: `serveStatic()` :: read function and serve a flat file
`serveRendered()` ::, parse a template to display `serveDynamic()` ::, connect to MySQL, apply data to a struct, and parse a template.

To take our requests and reroute, we'll change the `ServeHTTP` method for our `customRouter` struct to handle three regular expressions.

For the sake of brevity and clarity, we'll only be returning data on our three possible requests. Anything else will be ignored.

In a real-world scenario, we can take this approach to aggressively and proactively reject connections for requests we think are invalid. This would include spiders and nefarious bots and processes, which offer no real value as nonusers.

Serving pages

First up are our static pages. While we handled this the idiomatic way earlier, there exists the ability to rewrite our requests, better handle specific 404 error pages, and so on by using the `http.ServeFile` function, as shown in the following code:

```
path := r.URL.Path;

staticPatternString := "static/(.*)"
templatePatternString := "template/(.*)"
dynamicPatternString := "dynamic/(.*)"

staticPattern := regexp.MustCompile(staticPatternString)
templatePattern := regexp.MustCompile(templatePatternString)
dynamicDBPattern := regexp.MustCompile(dynamicPatternString)

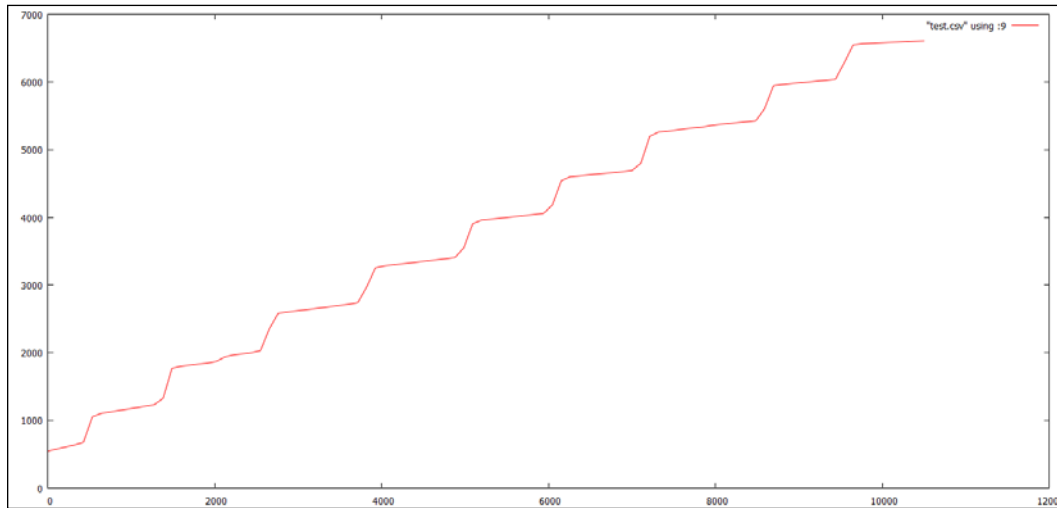
if staticPattern.MatchString(path) {
    page := staticPath + staticPattern.ReplaceAllString(path,
        "${1}") + ".html"

    http.ServeFile(rw, r, page)
}
```

Here, we simply relegate all requests starting with `/static/(.*)` to match the request in addition to the `.html` extension. In our case, we've named our test file (the 80 KB example file) `test.html`, so all requests to it will go to `/static/test`.

We've prepended this with `staticPath`, a constant defined upcode. In our case, it's `/var/www/`, but you'll want to modify it as necessary.

So, let's see what kind of overhead is imposed by introducing some regular expressions, as shown in the following graph:



How about that? Not only is there no overhead imposed, it appears that the `FileServer` functionality itself is heavier and slower than a distinct `FileServe()` call. Why is that? Among other reasons, not explicitly calling the file to open and serve imposes an additional OS call, one which can cascade as requests mount up at the expense of concurrency and performance.

Sometimes it's the little things



Other than strictly serving flat pages here, we're actually doing one other task per request using the following line of code:

```
fmt.Println(r.URL.Path)
```

While this ultimately may have no impact on your final performance, we should take care to avoid unnecessary logging or related activities that may impart seemingly minimal performance obstacles that become much larger ones at scale.

Parsing our template

In our next phase, we'll measure the impact of reading and parsing a template. To effectively match the previous tests, we'll take our HTML static file and impose some variables on it.

If you recall, our goal here is to mimic real-world scenarios as closely as possible. A real-world web server will certainly handle a lot of static file serving, but today, dynamic calls make up the vast bulk of web traffic.

Our data structure will resemble the simplest of data tables without having access to an actual database:

```
type WebPage struct {
    Title string
    Contents string
}
```

We'll want to take any data of this form and render a template with it. Remember that Go creates the notion of public or private variables through the syntactical sugar of capitalized (public) or lowercase (private) values.

If you find that the template fails to render but you're not given explicit errors in the console, check your variable naming. A private value that is called from an HTML (or text) template will cause rendering to stop at that point.

Now, we'll take that data and apply it to a template for any calls to a URL that begins with the `/(.*)` template. We could certainly do something more useful with the wildcard portion of that regular expression, so let's make it part of the title using the following code:

```
} else if templatePattern.MatchString(path) {

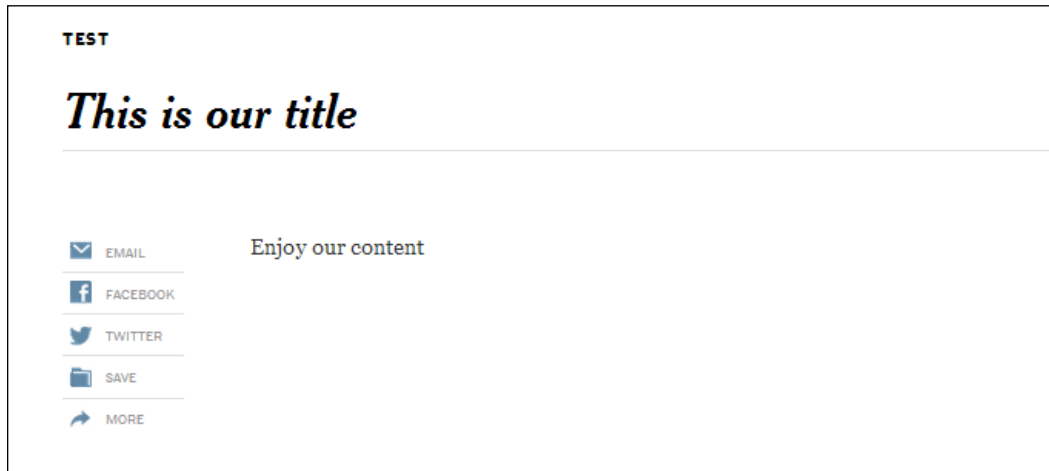
    urlVar := templatePattern.ReplaceAllString(path, "${1}")
    page := WebPage{ Title: "This is our URL: "+urlVar, Contents:
        "Enjoy our content" }
    tmp, _ := template.ParseFiles(staticPath+"template.html")
    tmp.Execute(rw, page)

}
```

Hitting `localhost:9000/template/hello` should render a template with a primary body of the following code:

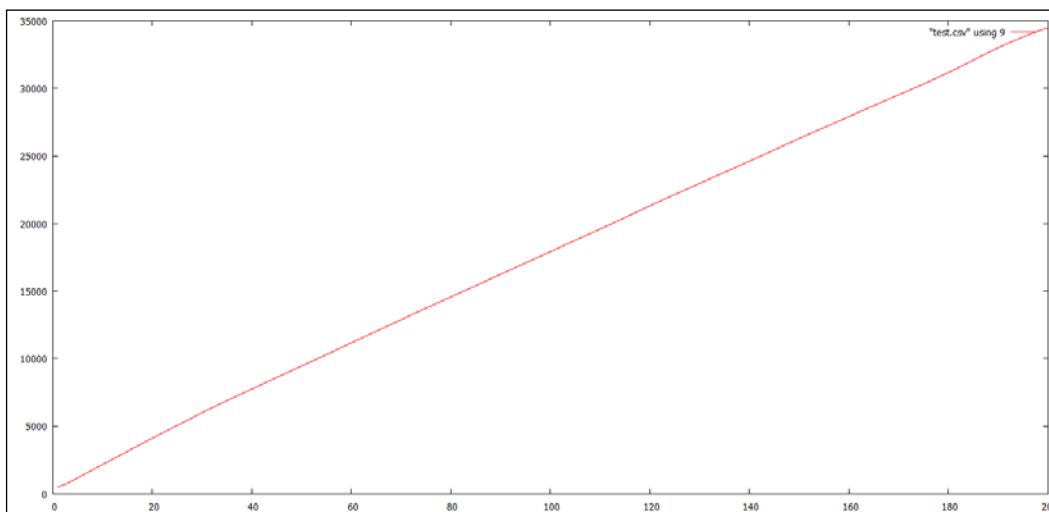
```
<h1>{{.Title}}</h1>
<p>{{.Contents}}</p>
```

We will do this with the following output:



One thing to note about templates is that they are not compiled; they remain dynamic. That is to say, if you create a renderable template and start your server, the template can be modified and the results are reflected.

This is noteworthy as a potential performance factor. Let's run our benchmarks again, with template rendering as the added complexity to our application and its architecture:



Yikes! What happened? We've gone from easily hitting 10,000 concurrent requests to barely handling 200.

To be fair, we introduced an intentional stumbling block, one not all that uncommon in the design of any given CMS.

You'll notice that we're calling the `template.ParseFiles()` method on every request. This is the sort of seemingly cheap call that can really add up when you start stacking the requests.

It may then make sense to move the file operations outside of the request handler, but we'll need to do more than that – to eliminate overhead and a blocking call, we need to set an internal cache for the requests.

Most importantly, all of our template creation and parsing should happen outside the actual request handler if you want to keep your server non-blocking, fast, and responsive. Here's another take:

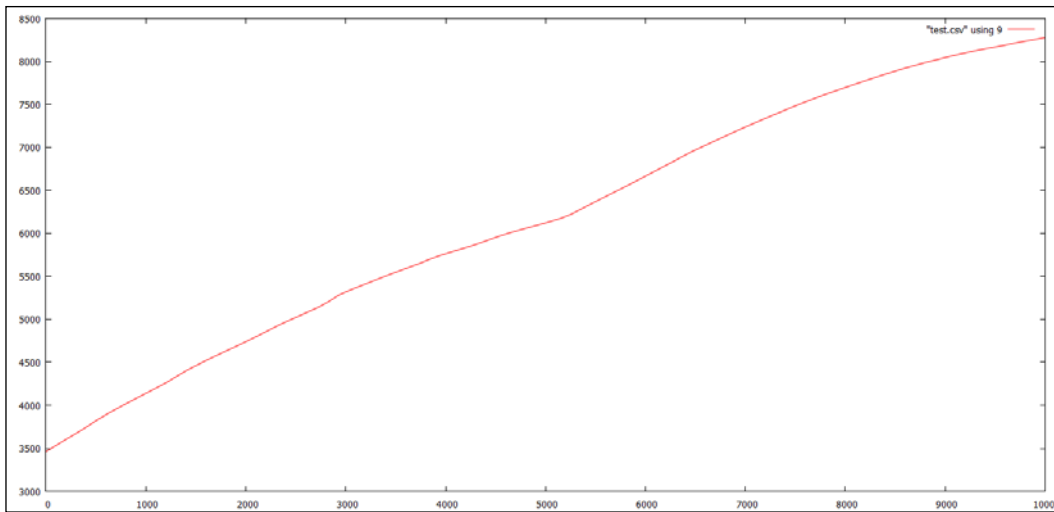
```
var customHTML string
var customTemplate template.Template
var page WebPage
var templateSet bool

func main() {
    var cr customRouter;
    fileName := staticPath + "template.html"
    cH,_ := ioutil.ReadFile(fileName)
    customHTML = string(cH[:])

    page := WebPage{ Title: "This is our URL: ", Contents: "Enjoy
        our content" }
    cT,_ := template.New("Hey").Parse(customHTML)
    customTemplate = *cT
```

Even though we're using the `Parse()` function prior to our request, we can still modify our URL-specific variables using the `Execute()` method, which does not carry the same overhead as `Parse()`.

When we move this outside of the `customRouter` struct's `ServeHTTP()` method, we're back in business. This is the kind of response we'll get with these changes:



External dependencies

Finally, we need to bring in our biggest potential bottleneck, which is the database. As mentioned earlier, we'll simulate random traffic by generating a random integer between 1 and 10,000 to specify the article we want.

Randomization isn't just useful on the frontend — we'll want to work around any query caching within MySQL itself to limit nonserver optimizations.

Connecting to MySQL

We can route our way through a custom connection to MySQL using native Go, but as is often the case, there are a few third-party packages that make this process far less painful. Given that the database here (and associated libraries) is tertiary to the primary exercise, we'll not be too concerned about the particulars here.

The two mature MySQL driver libraries are as follows:

- **Go-MySQL-Driver** (<https://github.com/go-sql-driver/mysql>)
- **MyMySQL** (<https://github.com/ziutek/mymysql>)

For this example, we'll go with the Go-MySQL-Driver. We'll quickly install it using the following command:

```
go get github.com/go-sql-driver/mysql
```

Both of these implement the core SQL database connectivity package in Go, which provides a standardized method to connect to a SQL source and iterate over rows.

One caveat is if you've never used the SQL package in Go but have in other languages – typically, in other languages, the notion of an `Open()` method implies an open connection. In Go, this simply creates the struct and relevant implemented methods for a database. This means that simply calling `Open()` on `sql.database` may not give you relevant connection errors such as username/password issues and so on.

One advantage of this (or disadvantage depending on your vantage point) is that connections to your database may not be left open between requests to your web server. The impact of opening and reopening connections is negligible in the grand scheme.

As we're utilizing a pseudo-random article request, we'll build a MySQL piggyback function to get an article by ID, as shown in the following code:

```
func getArticle(id int) WebPage {
    Database, err := sql.Open("mysql", "test:test@/master")
    if err != nil {
        fmt.Println("DB error!!!")
    }

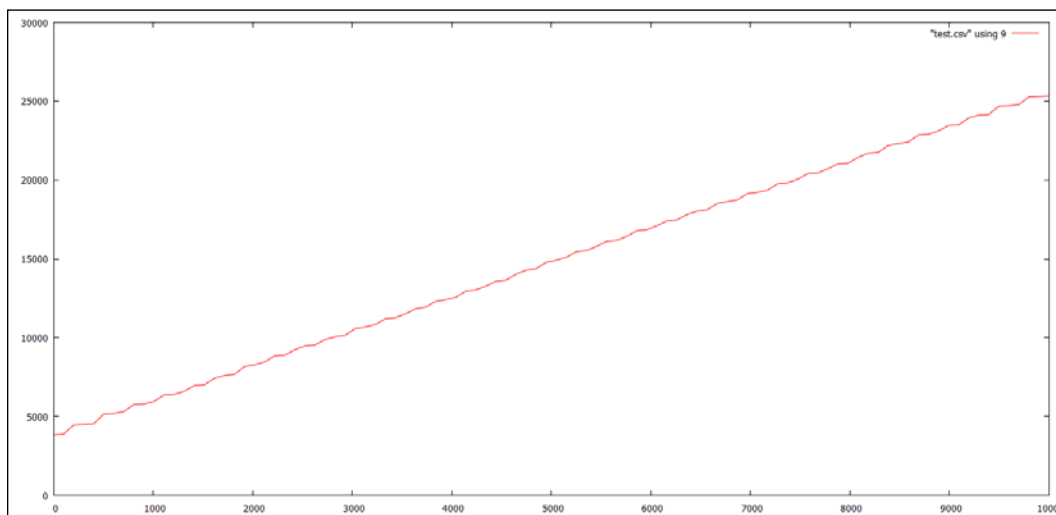
    var articleTitle string
    sqlQ := Database.QueryRow("SELECT article_title from articles
        where article_id=? LIMIT 1", 1).Scan(&articleTitle)
    switch {
        case sqlQ == sql.ErrNoRows:
            fmt.Printf("No rows!")
        case sqlQ != nil:
            fmt.Println(sqlQ)
        default:
    }

    wp := WebPage{}
    wp.Title = articleTitle
    return wp
}
```

We will then call the function directly from our `ServeHTTP()` method, as shown in the following code:

```
}else if dynamicDBPattern.MatchString(path) {  
    rand.Seed(9)  
    id := rand.Intn(10000)  
    page = getArticle(id)  
    customTemplate.Execute(rw, page)  
}
```

How did we do here? Take a look at the following graph:



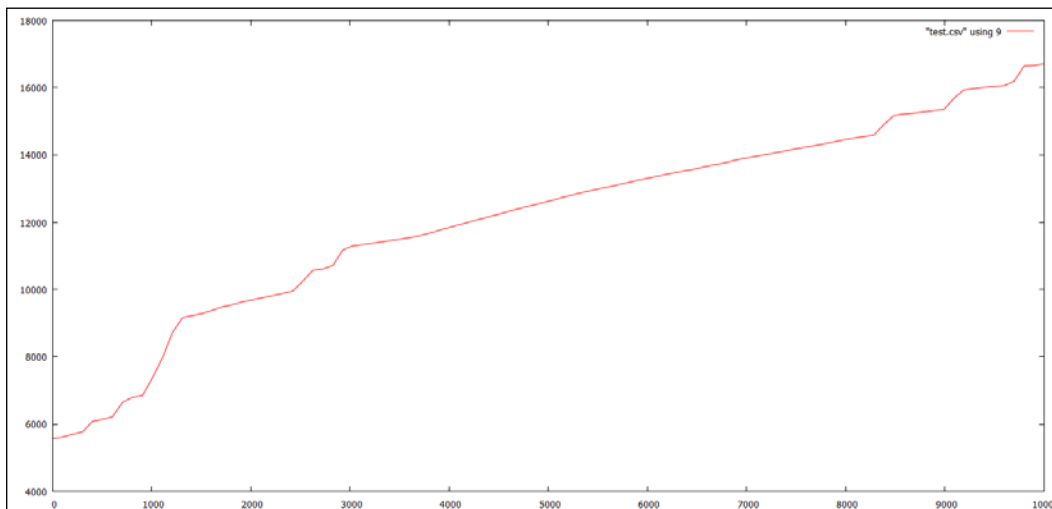
Slower, no doubt, but we held up to all 10,000 concurrent requests, entirely from uncached MySQL calls.

Given that we couldn't hit 1,000 concurrent requests with a default installation of Apache, this is nothing to sneeze at.

Multithreading and leveraging multiple cores

You may be wondering how performance may vary when invoking additional processor cores – as mentioned earlier, this can sometimes have an unexpected effect.

In this case, we should expect only improved performance in our dynamic requests and static requests. Any time the cost of context switching in the OS might outweigh the performance advantages of additional cores, we can see paradoxical performance degradation. In this case, we do not see this effect and instead see a relatively similar line, as shown in the following graph:



Exploring our web server

Our final web server is capable of serving static, template-rendered, and dynamic content well within the confines of the goal of 10,000 concurrent connections on even the most modest of hardware.

The code – much like the code in this book – can be considered a jumping-off point and will need refinement if put into production. This server lacks anything in the form of error handling but can ably serve valid requests without any issue. Let's take a look at the following server's code:

```
package main

import
(
```

```

"net/http"
"html/template"
"time"
"regexp"
"fmt"
"io/ioutil"
"database/sql"
"log"
"runtime"
_ "github.com/go-sql-driver/mysql"
)

```

Most of our imports here are fairly standard, but note the MySQL line that is called solely for its side effects as a database/SQL driver:

```
const staticPath string = "static/"
```

The relative `static/` path is where we'll look for any file requests—as mentioned earlier, this does no additional error handling, but the `net/http` package itself will deliver 404 errors should a request to a nonexistent file hit it:

```

type WebPage struct {

    Title string
    Contents string
    Connection *sql.DB

}

```

Our `WebPage` type represents the final output page before template rendering. It can be filled with static content or populated by data source, as shown in the following code:

```

type customRouter struct {

}

func serveDynamic() {

}

func serveRendered() {

}

func serveStatic() {

}

```

Use these if you choose to extend the web app – this makes the code cleaner and removes a lot of the cruft in the `ServeHTTP` section, as shown in the following code:

```
func (customRouter) ServeHTTP(rw http.ResponseWriter, r
    *http.Request) {
    path := r.URL.Path;

    staticPatternString := "static/(.*)"
    templatePatternString := "template/(.*)"
    dynamicPatternString := "dynamic/(.*)"

    staticPattern := regexp.MustCompile(staticPatternString)
    templatePattern := regexp.MustCompile(templatePatternString)
    dynamicDBPattern := regexp.MustCompile(dynamicPatternString)

    if staticPattern.MatchString(path) {
        serveStatic()
        page := staticPath + staticPattern.ReplaceAllString(path,
            "${1}") + ".html"
        http.ServeFile(rw, r, page)
    } else if templatePattern.MatchString(path) {

        serveRendered()
        urlVar := templatePattern.ReplaceAllString(path, "${1}")

        page.Title = "This is our URL: " + urlVar
        customTemplate.Execute(rw, page)

    } else if dynamicDBPattern.MatchString(path) {

        serveDynamic()
        page = getArticle(1)
        customTemplate.Execute(rw, page)
    }
}
```

All of our routing here is based on regular expression pattern matching. There are a lot of ways you can do this, but `regexp` gives us a lot of flexibility. The only time you may consider simplifying this is if you have so many potential patterns that it could cause a performance hit—and this means thousands. The popular web servers, Nginx and Apache, handle a lot of their configurable routing through regular expressions, so it's fairly safe territory:

```
func gobble(s []byte) {

}
```

Go is notoriously cranky about unused variables, and while this isn't always the best practice, you will end up, at some point, with a function that does nothing specific with data but keeps the compiler happy. For production, this is not the way you'd want to handle such data.

```
var customHTML string
var customTemplate template.Template
var page WebPage
var templateSet bool
var Database sql.DB

func getArticle(id int) WebPage {
    Database, err := sql.Open("mysql", "test:test@/master")
    if err != nil {
        fmt.Println("DB error!")
    }

    var articleTitle string
    sqlQ := Database.QueryRow("SELECT article_title from articles
        WHERE article_id=? LIMIT 1", id).Scan(&articleTitle)
    switch {
        case sqlQ == sql.ErrNoRows:
            fmt.Printf("No rows!")
        case sqlQ != nil:
            fmt.Println(sqlQ)
        default:

    }

    wp := WebPage{}
    wp.Title = articleTitle
    return wp
}
```


Our `getArticle` function demonstrates how you can interact with the database/sql package at a very basic level. Here, we open a connection and query a single row with the `QueryRow()` function. There also exists the `Query` command, which is also usually a `SELECT` command but one that could return more than a single row.

```
func main() {

    runtime.GOMAXPROCS(4)

    var cr customRouter;

    fileName := staticPath + "template.html"
    cH,_ := ioutil.ReadFile(fileName)
    customHTML = string(cH[:])

    page := WebPage{ Title: "This is our URL: ", Contents: "Enjoy
        our content" }
    cT,_ := template.New("Hey").Parse(customHTML)
    customTemplate = *cT

    gobble(cH)
    log.Println(page)
    fmt.Println(customTemplate)

    server := &http.Server {
        Addr: ":9000",
        Handler:cr,
        ReadTimeout: 10 * time.Second,
        WriteTimeout: 10 * time.Second,
        MaxHeaderBytes: 1 << 20,
    }

    server.ListenAndServe()

}
```

Our main function sets up the server, builds a default `WebPage` and `customRouter`, and starts listening on port 9000.

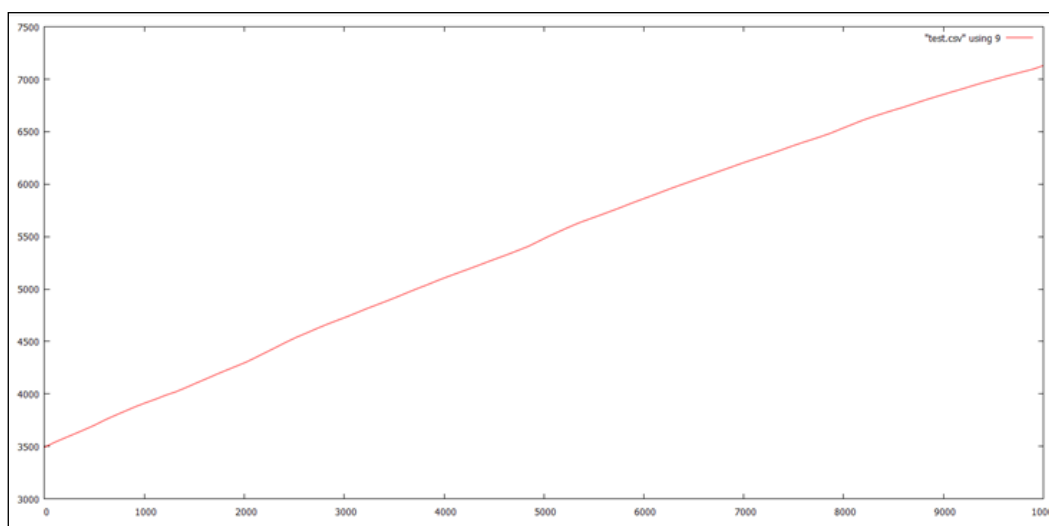
Timing out and moving on

One thing we did not focus on in our server is the notion of lingering connection mitigation. The reason we didn't worry much about it is because we were able to hit 10,000 concurrent connections in all three approaches without too much issue, strictly by utilizing Go's powerful built-in concurrency features.

Particularly when working with third-party or external applications and services, it's important to know that we can and should be prepared to call it quits on a connection (if our application design permits it).

Note the custom server implementation and two notes-specific properties: `ReadTimeout` and `WriteTimeout`. These allow us to handle this use case precisely.

In our example, this is set to an absurdly high 10 seconds. For a request to be received, processed, and sent, up to 20 seconds can transpire. This is an eternity in the Web world and has the potential to cripple our application. So, what does our C10K look like with 1 second on each end? Let's take a look at the following graph:



Here, we've saved nearly 5 seconds off the tail end of our highest volume of concurrent requests, almost certainly at the expense of complete responses to each.

It's up to you to decide how long it's acceptable to keep slow-running connections, but it's another tool in the arsenal to keep your server swift and responsive.

There will always be a tradeoff when you decide to kill a connection—too early and you'll have a bevy of complaints about a nonresponsive or error-prone server; too late and you'll be unable to cope with the connection volume programmatically. This is one of those considerations that will require QA and hard data.

Summary

The C10K problem may seem like a relic today, but the call to action was symptomatic of the type of approaches to systems' applications that were primarily employed prior to the rapid expansion of concurrent languages and application design.

Just 15 years ago, this seemed a largely insurmountable problem facing systems and server developers worldwide; now, it's handled with only minor tweaking and consideration by a server designer.

Go makes it easy to get there (with a little effort), but reaching 10,000 (or 100,000 or even 1,000,000) concurrent connections is only half the battle. We must know what to do when problems arise, how to seek out maximum performance and responsiveness out of our servers, and how to structure our external dependencies such that they do not create roadblocks.

In our next chapter, we'll look at squeezing even more performance out of our concurrent applications by testing some distributed computing patterns and best utilizing memory management.

7

Performance and Scalability

To build a high-powered web server in Go with just a few hundred lines of code, you should be quite aware of how concurrent Go provides us with exceptional tools for performance and stability out of the box.

Our example in *Chapter 6, C10K – A Non-blocking Web Server in Go*, also showed how imposing blocking code arbitrarily or inadvertently into our code can introduce some serious bottlenecks and quickly torpedo any plans to extend or scale your application.

What we'll look at in this chapter are a few ways that can better prepare us to take our concurrent application and ensure that it's able to continuously scale in the future and that it is capable of being expanded in scope, design, and/or capacity.

We'll expand a bit on **pprof**, the CPU profiling tool we looked at briefly in previous chapters, as a way to elucidate the way our Go code is compiled and to locate possible unintended bottlenecks.

Then we'll expand into distributed Go and into ways to offer some performance-enhancing parallel-computing concepts to our applications. We'll also look at the Google App Engine, and at how you can utilize it for your Go-based applications to ensure scalability is placed in the hands of one of the most reliable hosting infrastructures in the world.

Lastly, we'll look at memory utilization, preservation, and how Google's garbage collector works (and sometimes doesn't). We'll finally delve a bit deeper into using memory caching to keep data consistent as well as less ephemeral, and we will also see how that dovetails with distributed computing in general.

High performance in Go

Up to this point, we've talked about some of the tools we can use to help discover slowdowns, leaks, and inefficient looping.

Go's compiler and its built-in deadlock detector keep us from making the kind of mistake that's common and difficult to detect in other languages.

We've run time-based benchmarks based on specific changes to our concurrency patterns, which can help us design our application using different methodologies to improve overall execution speed and performance.

Getting deeper into pprof

The pprof tool was first encountered in *Chapter 5, Locks, Blocks, and Better Channels*, and if it still feels a bit cryptic, that's totally understandable. What pprof shows you in export is a **call graph**, and we can use this to help identify issues with loops or expensive calls on the heap. These include memory leaks and processor-intensive methods that can be optimized.

One of the best ways to demonstrate how something like this works is to build something that doesn't. Or at least something that doesn't work the way it should.

You might be thinking that a language with garbage collection might be immune to these kinds of memory issues, but there are always ways to hide mistakes that can lead to memory leakage. If the GC can't find it, it can sometimes be a real pain to do so yourself, leading to a lot of — often feckless — debugging.

To be fair, what constitutes a memory leak is sometimes debated among computer science members and experts. A program that continuously consumes RAM may not be leaking memory by technical definition if the application itself could re-access any given pointers. But that's largely irrelevant when you have a program that crashes and burns after consuming memory like an elephant at a buffet.

The basic premise of creating a memory leak in a garbage-collected language relies on hiding the allocation from the compiler — indeed, any language in which you can access and utilize memory directly provides a mechanism for introducing leaks.

We'll review a bit more about garbage collection and Go's implementation later in this chapter.

So how does a tool like pprof help? Very simply put, by showing you **where** your memory and CPU utilization goes.

Let's first design a very obvious CPU hog as follows to see how pprof highlights this for us:

```
package main

import (
    "os"
    "flag"
    "fmt"
    "runtime/pprof"
)

const TESTLENGTH = 100000
type CPUHog struct {
    longByte []byte
}

func makeLongByte() []byte {
    longByte := make([]byte, TESTLENGTH)

    for i:= 0; i < TESTLENGTH; i++ {
        longByte[i] = byte(i)
    }
    return longByte
}

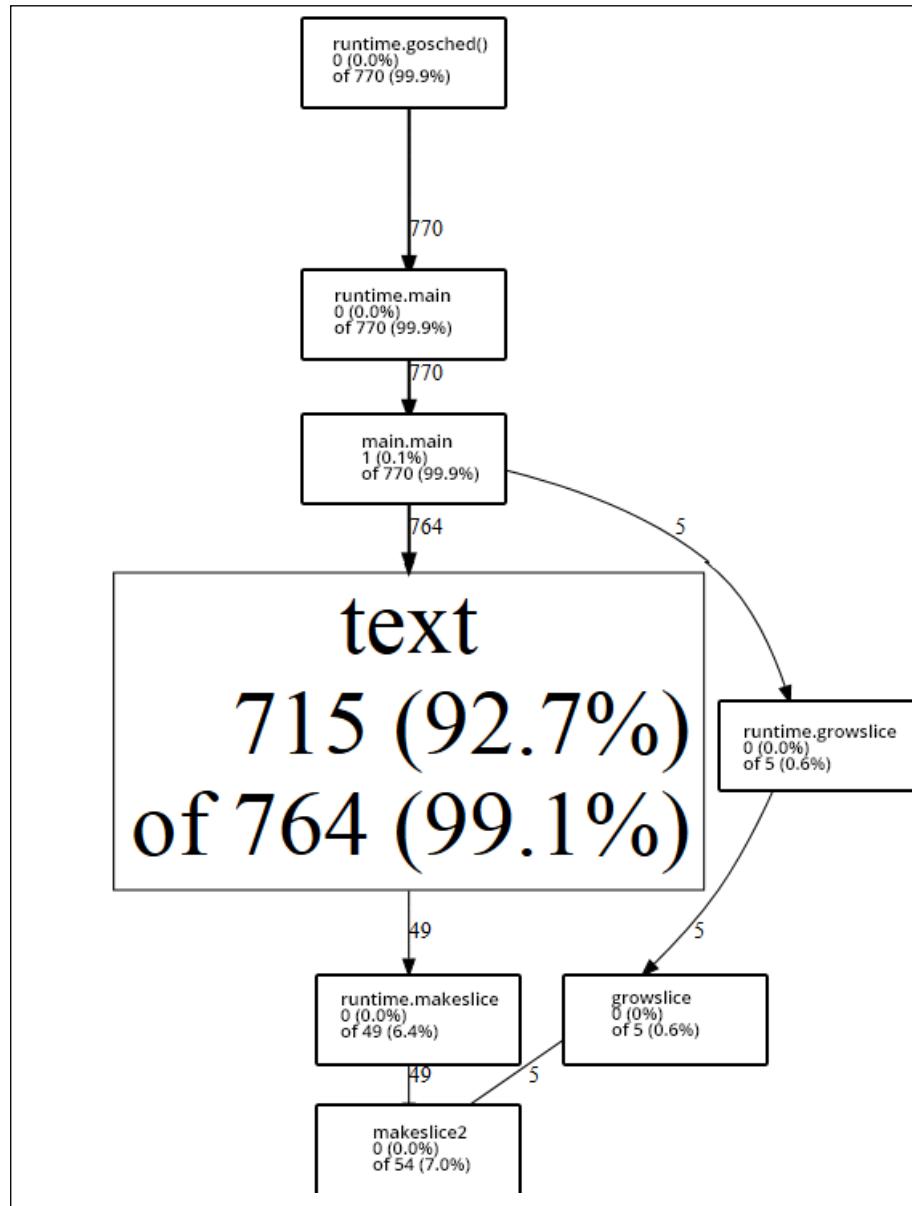
var profile = flag.String("cpuprofile", "", "output pprof data to
    file")

func main() {
    var CPUHogs []CPUHog

    flag.Parse()
    if *profile != "" {
        flag, err := os.Create(*profile)
        if err != nil {
            fmt.Println("Could not create profile", err)
        }
        pprof.StartCPUProfile(flag)
        defer pprof.StopCPUProfile()
    }

    for i := 0; i < TESTLENGTH; i++ {
        hog := CPUHog{}
        hog.longByte = makeLongByte()
        _ = append(CPUHogs, hog)
    }
}
```

The output of the preceding code is shown in the following diagram:



In this case, we know where our stack resource allocation is going, because we willfully introduced the loop (and the loop within that loop).

Imagine that we didn't intentionally do that and had to locate resource hogs. In this case, pprof makes this pretty easy, showing us the creation and memory allocation of simple strings comprising the majority of our samples.

We can modify this slightly to see the changes in the pprof output. In an effort to allocate more and more memory to see whether we can vary the pprof output, we might consider heavier types and more memory.

The easiest way to accomplish that is to create a slice of a new type that includes a significant amount of these heavier types such as `int64`. We're blessed with Go: in that, we aren't prone to common C issues such as buffer overflows and memory protection and management, but this makes debugging a little trickier when we cannot intentionally break the memory management system.

The unsafe package

Despite the built-in memory protection provided, there is still another interesting tool provided by Go: the **unsafe** package. As per Go's documentation:

Package unsafe contains operations that step around the type safety of Go programs.



This might seem like a curious library to include—indeed, while many low-level languages allow you to shoot your foot off, it's fairly unusual to provide a segregated language.

Later in this chapter, we'll examine `unsafe.Pointer`, which allows you to read and write to arbitrary bits of memory allocation. This is obviously extraordinarily dangerous (or useful and nefarious, depending on your goal) functionality that you would generally try to avoid in any development language, but it does allow us to debug and understand our programs and the Go garbage collector a bit better.

So to increase our memory usage, let's switch our string allocation as follows, for random type allocation, specifically for our new struct `MemoryHog`:

```
type MemoryHog struct {
    a,b,c,d,e,f,g int64
    h,i,j,k,l,m,n float64
    longByte []byte
}
```

There's obviously nothing preventing us from extending this into some ludicrously large set of slices, huge arrays of `int64`s, and so on. But our primary goal is solely to change the output of pprof so that we can identify movement in the call graph's samples and its effect on our stack/heap profiles.

Our arbitrarily expensive code looks as follows:

```
type MemoryHog struct {
    a,b,c,d,e,f,g int64
    h,i,j,k,l,m,n float64
    longByte []byte
}

func makeMemoryHog() []MemoryHog {

    memoryHogs := make([]MemoryHog, TESTLENGTH)

    for i:= 0; i < TESTLENGTH; i++ {
        m := MemoryHog{}
        _ = append(memoryHogs,m)
    }

    return memoryHogs
}

var profile = flag.String("cpuprofile", "", "output pprof data to
file")

func main() {
    var CPUHogs []CPUHog

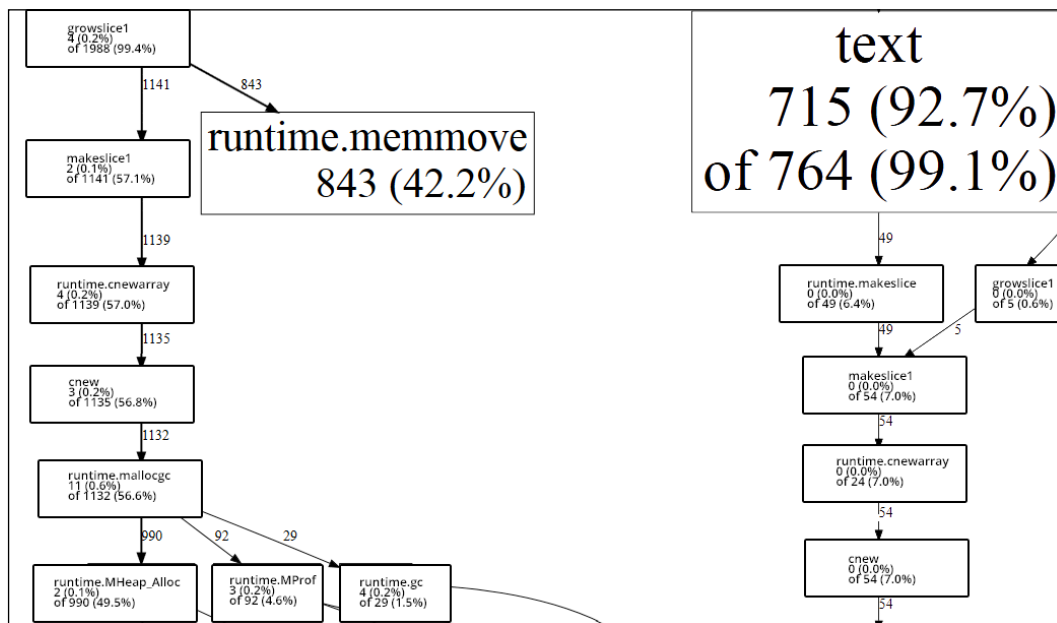
    flag.Parse()
    if *profile != "" {
        flag,err := os.Create(*profile)
        if err != nil {
            fmt.Println("Could not create profile",err)
        }
        pprof.StartCPUProfile(flag)
        defer pprof.StopCPUProfile()

    }

    for i := 0; i < TESTLENGTH; i++ {
        hog := CPUHog{}
        hog.mHog = makeMemoryHog()
        _ = append(CPUHogs,hog)
    }
}
```

With this in place, our CPU consumption remains about the same (due to the looping mechanism remaining largely unchanged), but our memory allocation has increased – unsurprisingly – by about 900 percent. It's unlikely that you will precisely duplicate these results, but the general trend of a small change leading to a major difference in resource allocation is reproducible. Note that memory utilization reporting is possible with pprof, but it's not what we're doing here; the memory utilization observations here happened outside of pprof.

If we took the extreme approach suggested previously – to create absurdly large properties for our struct – we could carry that out even further, but let's see what the aggregate impact is on our CPU profile on execution. The impact is shown in the following diagram:



On the left-hand side, we have our new allocation approach, which invokes our larger struct instead of an array of strings. On the right-hand side, we have our initial application.

A pretty dramatic flux, don't you think? While neither of these programs is wrong in design, we can easily toggle our methodologies to see where resources are going and discern how we can reduce their consumption.

Parallelism's and concurrency's impact on I/O pprof

One issue you'll likely run into pretty quickly when using pprof is when you've written a script or application that is especially bound to efficient runtime performance. This happens most frequently when your program executes too quickly to properly profile.

A related issue involves network applications that require connections to profile; in this case, you can simulate traffic either in-program or externally to allow proper profiling.

We can demonstrate this easily by replicating something like the preceding example with goroutines as follows:

```
const TESTLENGTH = 20000

type DataType struct {
    a,b,c,d,e,f,g int64
    longByte []byte
}

func (dt DataType) init() {

}

var profile = flag.String("cpuprofile", "", "output pprof data to file")

func main() {

    flag.Parse()
    if *profile != "" {
        flag,err := os.Create(*profile)
        if err != nil {
            fmt.Println("Could not create profile",err)
        }
        pprof.StartCPUProfile(flag)
        defer pprof.StopCPUProfile()
    }
```

```

var wg sync.WaitGroup

numCPU := runtime.NumCPU()
runtime.GOMAXPROCS(numCPU)

wg.Add(TESTLENGTH)

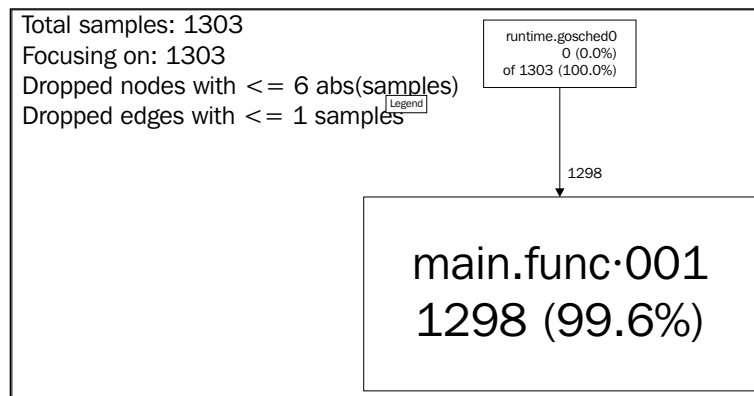
for i := 0; i < TESTLENGTH; i++ {
    go func() {
        for y := 0; y < TESTLENGTH; y++ {
            dT := DataType{}
            dT.init()
        }
        wg.Done()
    }()
}

wg.Wait()

fmt.Println("Complete.")
}

```

The following diagram shows the pprof output of the preceding code:



It's not nearly as informative, is it?

If we want to get something more valuable about the stack trace of our goroutines, Go—as usual—provides some additional functionality.

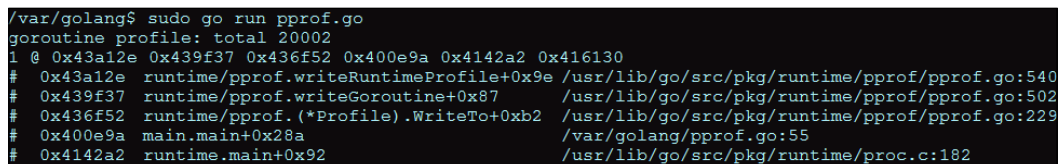
In the runtime package, there is a function and a method that allow us to access and utilize the stack traces of our goroutines:

- `runtime.Lookup`: This function returns a profile based on name
- `runtime.WriteTo`: This method sends the snapshot to the I/O writer

If we add the following line to our program, we won't see the output in the pprof Go tool, but we can get a detailed analysis of our goroutines in the console.

```
pprof.Lookup("goroutine").WriteTo(os.Stdout, 1)
```

The previous code line gives us some more of the abstract goroutine memory location information and package detail, which will look something like the following screenshot:



```
/var/golang$ sudo go run pprof.go
goroutine profile: total 20002
1 @ 0x43a12e 0x439f37 0x436f52 0x400e9a 0x4142a2 0x416130
# 0x43a12e runtime/pprof.writeRuntimeProfile+0x9e /usr/lib/go/src/pkg/runtime/pprof/pprof.go:540
# 0x439f37 runtime/pprof.writeGoroutine+0x87 /usr/lib/go/src/pkg/runtime/pprof/pprof.go:502
# 0x436f52 runtime/pprof.(*Profile).WriteTo+0xb2 /usr/lib/go/src/pkg/runtime/pprof/pprof.go:229
# 0x400e9a main.main+0x28a /var/golang/pprof.go:55
# 0x4142a2 runtime.main+0x92 /usr/lib/go/src/pkg/runtime/proc.c:182
```

But an even faster way to get this output is by utilizing the `http/pprof` tool, which keeps the results of our application active via a separate server. We've gone with port 6000 here as shown in the following code, though you can modify this as necessary:

```
go func() {
    log.Println(http.ListenAndServe("localhost:6000", nil))
}()
```

While you cannot get an SVG output of the goroutine stack call, you can see it live in your browser by going to `http://localhost:6060/debug/pprof/goroutine?debug=1`.

Using the App Engine

While not right for every project, Google's App Engine can open up a world of scalability when it comes to concurrent applications, without the hassle of VM provisioning, reboots, monitoring, and so on.

The App Engine is not entirely dissimilar to Amazon Web Services, DigitalOcean, and the ilk, except for the fact that you do not need to necessarily involve yourself in the minute details of direct server setup and maintenance. All of them provide a single spot to acquire and utilize virtual computing resources for your applications.

Rather, it can be a more abstract environment within Google's architecture with which to house and run your code in a number of languages, including — no surprise here — the Go language itself.

While large-scale apps will cost you, Google provides a free tier with reasonable quotas for experimentation and small applications.

The benefits as they relate to scalability here are two-fold: you're not responsible for ensuring uptime on the instances as you would be in an AWS or DigitalOcean scenario. Who else but Google will have not only the architecture to support anything you can throw at it, but also have the fastest updates to the Go core itself?

There are some obvious limitations here that coincide with the advantages, of course, including the fact that your core application will be available exclusively via `http` (although it will have access to plenty of other services).



To deploy apps to the App Engine, you'll need the SDK for Go, available for Mac OS X, Linux, and Windows, at https://developers.google.com/appengine/downloads#Google_App_Engine_SDK_for_Go.

Once you've installed the SDK, the changes you'll need to make to your code are minor — the most noteworthy point is that for most cases, your `Go` tool command will be supplanted by `goapp`, which handles serving your application locally and then deploying it.

Distributed Go

We've certainly covered a lot about concurrent and parallel Go, but one of the biggest infrastructure challenges for developers and system architects today has to do with cooperative computing.

Some of the applications and designs that we've mentioned previously scale from parallelism to distributed computing.

Memcache(d) is a form of in-memory caching, which can be used as a queue among several systems.

Our master-slave and producer-consumer models we presented in *Chapter 4, Data Integrity in an Application*, have more to do with distributed computing than single-machine programming in Go, which manages concurrency idiomatically. These models are typical concurrency models in many languages, but can be scaled to help us design distributed systems as well, utilizing not just many cores and vast resources but also redundancy.

The basic premise of distributed computing is to share, spread, and best absorb the various burdens of any given application across many systems. This not only improves performance on aggregate, but provides some sense of redundancy for the system itself.

This all comes at some cost though, which are as follows:

- Potential for network latency
- Creating slowdowns in communication and in application execution
- Overall increase in complexity both in design and in maintenance
- Potential for security issues at various nodes along the distributed route(s)
- Possible added cost due to bandwidth considerations

This is all to say, simply, that while building a distributed system can provide great benefits to a large-scale application that utilizes concurrency and ensures data consistency, it's by no means right for every example.

Types of topologies

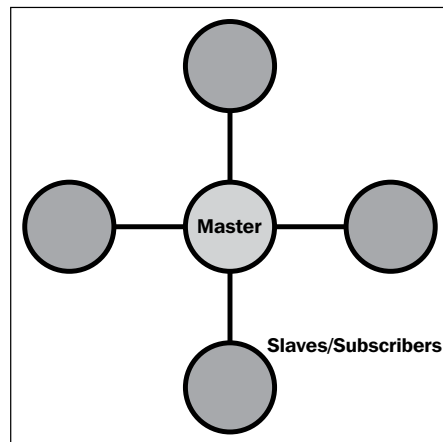
Distributed computing recognizes a slew of logical topologies for distributed design. Topology is an apt metaphor, because the positioning and logic of the systems involved can often represent physical topology.

Out of the box, not all of the accepted topologies apply to Go. When we design concurrent, distributed applications using Go, we'll generally rely on a few of the simpler designs, which are as follows.

Type 1 – star

The star topology (or at least this particular form of it), resembles our master-slave or producer-consumer models as outlined previously.

The primary method of data passing involves using the master as a message-passing conduit; in other words, all requests and commands are coordinated by a single instance, which uses some routing method to pass messages. The following diagram shows the star topology:



We can actually very quickly design a goroutine-based system for this. The following code is solely the master's (or distributed destination's) code and lacks any sort of security considerations, but shows how we can parlay network calls to goroutines:

```
package main

import (
    "fmt"
    "net"
)
```

Our standard, basic libraries are defined as follows:

```
type Subscriber struct {
    Address net.Addr
    Connection net.Conn
    do chan Task
}

type Task struct {
    name string
}
```


These are the two custom types we'll use here. A `Subscriber` type is any distributed helper that comes into the fray, and a `Task` type represents any given distributable task. We've left that undefined here because it's not the primary goal of demonstration, but you could ostensibly have `Task` do anything by communicating standardized commands across the TCP connection. The `Subscriber` type is defined as follows:

```
var SubscriberCount int
var Subscribers []Subscriber
var CurrentSubscriber int
var taskChannel chan Task

func (sb Subscriber) awaitTask() {
    select {
        case t := <-sb.do:
            fmt.Println(t.name, "assigned")
    }
}

func serverListen (listener net.Listener) {
    for {
        conn, _ := listener.Accept()

        SubscriberCount++

        subscriber := Subscriber{ Address: conn.RemoteAddr(),
            Connection: conn }
        subscriber.do = make(chan Task)
        subscriber.awaitTask()
        _ = append(Subscribers, subscriber)
    }
}

func doTask() {
    for {
        select {
            case task := <-taskChannel:
                fmt.Println(task.name, "invoked")
                Subscribers[CurrentSubscriber].do <- task
                if (CurrentSubscriber+1) > SubscriberCount {
                    CurrentSubscriber = 0
                } else {
```

```
        CurrentSubscriber++
    }
}

}

}

func main() {

    destinationStatus := make(chan int)

    SubscriberCount = 0
    CurrentSubscriber = 0

    taskChannel = make(chan Task)

    listener, err := net.Listen("tcp", ":9000")
    if err != nil {
        fmt.Println ("Could not start server!",err)
    }
    go serverListen(listener)
    go doTask()

    <-destinationStatus
}
```

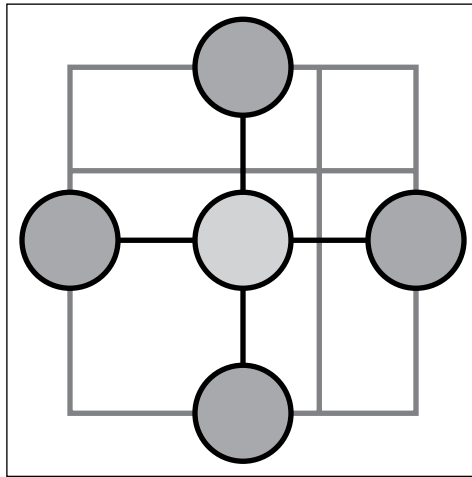
This essentially treats every connection as a new `Subscriber`, which gets its own channel based on its index. This master server then iterates through existing `Subscriber` connections using the following very basic round-robin approach:

```
if (CurrentSubscriber+1) > SubscriberCount {
    CurrentSubscriber = 0
}else {
    CurrentSubscriber++
}
```

As mentioned previously, this lacks any sort of security model, which means that any connection to port 9000 would become a `Subscriber` and could get network messages assigned to it (and ostensibly could invoke new messages too). But you may have noticed an even bigger omission: this distributed application doesn't do anything. Indeed, this is just a model for assignment and management of subscribers. Right now, it doesn't have any path of action, but we'll change that later in this chapter.

Type 2 – mesh

The mesh is very similar to the star with one major difference: each node is able to communicate not just through the master, but also directly with other nodes as well. This is also known as a **complete graph**. The following diagram shows a mesh topology:



For practical purposes, the master must still handle assignments and pass connections back to the various nodes.

This is actually not particularly difficult to add through the following simple modification of our previous server code:

```
func serverListen (listener net.Listener) {
    for {
        conn, _ := listener.Accept()

        SubscriberCount++

        subscriber := Subscriber{ Address: conn.RemoteAddr(),
                                   Connection: conn }
        subscriber.awaitTask()
        _ = append(Subscribers, subscriber)
        broadcast()
    }
}
```

Then, we add the following corresponding `broadcast` function to share all available connections to all other connections:

```
func broadcast() {
    for i:= range Subscribers {
        for j:= range Subscribers {
            Subscribers[i].Connection.Write
                ([]byte("Subscriber:", Subscriber[j].Address))
        }
    }
}
```

The Publish and Subscribe model

In both the previous topologies, we've replicated a Publish and Subscribe model with a central/master handling delivery. Unlike in a single-system, concurrent pattern, we lack the ability to use channels directly across separate machines (unless we use something like Go's Circuit as described in *Chapter 4, Data Integrity in an Application*).

Without direct programmatic access to send and receive actual commands, we rely on some form of API. In the previous examples, there is no actual task being sent or executed, but how could we do this?

Obviously, to create tasks that can be formalized into non-code transmission, we'll need a form of API. We can do this one of two ways: serialization of commands, ideally via JSONDirect transmission, and execution of code.

As we'll always be dealing with compiled code, the serialization of commands option might seem like you couldn't include Go code itself. This isn't exactly true, but passing full code in any language is fairly high on lists of security concerns.

But let's look at two ways of sending data via API in a task by removing a URL from a slice of URLs for retrieval. We'll first need to initialize that array in our `main` function as shown in the following code:

```
type URL struct {
    URI string
    Status int
    Assigned Subscriber
    SubscriberID int
}
```

Every URL in our array will include the URI, its status, and the subscriber address to which it's been assigned. We'll formalize the status points as 0 for unassigned, 1 for assigned and waiting, and 2 for assigned and complete.

Remember our `CurrentSubscriber` iterator? That represents the next-in-line round robin assignment which will fulfill the `SubscriberID` value for our URL struct.

Next, we'll create an arbitrary array of URLs that will represent our overall job here. Some suspension of incredulity may be necessary to assume that the retrieval of four URLs should require any distributed system; in reality, this would introduce significant slowdown by virtue of network transmission. We've handled this in a purely single-system, concurrent application before:

```
URLs = []URL{ {Status:0,URL:"http://golang.org/"},
               {Status:0,URL:"http://play.golang.org/"},
               {Status:0,URL:"http://golang.org/doc/"},
               {Status:0,URL:"http://blog.golang.org/" } }
```

Serialized data

In our first option in the API, we'll send and receive serialized data in JSON. Our master will be responsible for formalizing its command and associated data. In this case, we'll want to transmit a few things: what to do (in this case, retrieve) with the relevant data, what the response should be when it is complete, and how to address errors.

We can represent this in a custom struct as follows:

```
type Assignment struct {
    command string
    data string
    successResponse string
    errorResponse string
}
...
asmnt := Assignment{command:"process",
    url:"http://www.golang.org",successResponse:"success",
    errorResponse:"error"}
json, _ := json.Marshal(asmnt)
send(string(json))
```

Remote code execution

The remote code execution option is not necessarily separate from serialization of commands, but instead of structured and interpreted formatted responses, the payload could be code that will be run via a system command.

As an example, code from any language could be passed through the network and executed from a shell or from a syscall library in another language, like the following Python example:

```
from subprocess import call
call([remoteCode])
```

The disadvantages to this approach are many: it introduces serious security issues and makes error detection within your client nearly impossible.

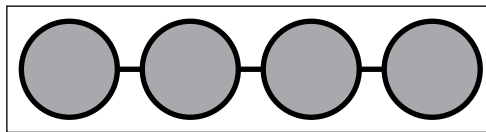
The advantages are you do not need to come up with a specific format and interpreter for responses as well as potential speed improvements. You can also offload the response code to another external process in any number of languages.

In most cases, serialization of commands is far preferable over the remote code execution option.

Other topologies

There exist quite a few topology types that are more complicated to manage as part of a messaging queue.

The following diagram shows the bus topology:

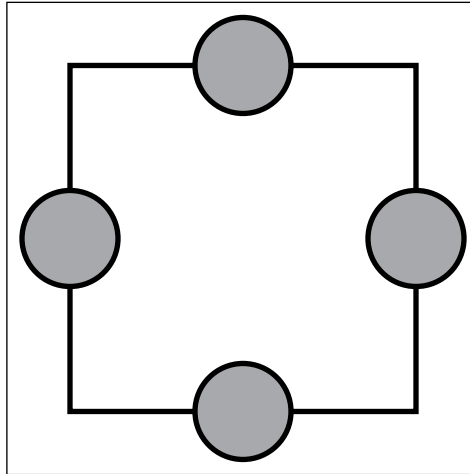


The bus topology network is a unidirectional transmission system. For our purposes, it's neither particularly useful nor easily managed, as each added node needs to announce its availability, accept listener responsibility, and be ready to cede that responsibility when a new node joins.

The advantage of a bus is quick scalability. This comes with serious disadvantages though: lack of redundancy and single point of failure.

Even with a more complex topology, there will always be some issue with potentially losing a valuable cog in the system; at this level of modular redundancy, some additional steps will be necessary to have an always-available system, including automatic double or triple node replication and failovers. That's a bit more than we'll get into here, but it's important to note that the risk will be there in any event, although it would be a little more vulnerable with a topology like the bus.

The following diagram shows the ring topology:



The ring topology looks similar to our mesh topology, but lacks a master. It essentially requires the same communication process (announce and listen) as does a bus. Note one significant difference: instead of a single listener, communication can happen between any node without the master.

This simply means that all nodes must both listen and announce their presence to other nodes.

Message Passing Interface

There exists a slightly more formalized version of what we built previously, called Message Passing Interface. MPI was borne from early 1990s academia as a standard for distributed communication.

Originally written with FORTRAN and C in mind, it is still a protocol, so it's largely language agnostic.

MPI allows the management of topology above and beyond the basic topologies we were able to build for a resource management system, including not only the line and ring but also the common bus topology.

For the most part, MPI is used by the scientific community; it is a highly concurrent and analogous method for building large-scale distributed systems. Point-to-point operations are more rigorously defined with error handling, retries, and dynamic spawning of processes all built in.

Our previous basic examples lend no prioritization to processors, for example, and this is a core effect of MPI.

There is no official implementation of MPI for Go, but as there exists one for both C and C++, it's entirely possible to interface with it through that.



There is also a simple and incomplete binding written in Go by Marcus Thierfelder that you can experiment with. It is available at <https://github.com/marcusthierfelder/mpi>.

You can read more about and install OpenMPI from <http://www.open-mpi.org/>.

Also you can read more about MPI and MPICH implementations at <http://www.mpich.org/>.

Some helpful libraries

There's little doubt that Go provides some of the best ancillary tools available to any compiled language out there. Compiling to native code on a myriad of systems, deadlock detection, pprof, fmt, and more allow you to not just build high-performance applications, but also test them and format them.

This hasn't stopped the community from developing other tools that can be used for debugging or aiding your concurrent and/or distributed code. We'll take a look at a few great tools that may prove worthy of inclusion in your app, particularly if it's highly visible or performance critical.

Nitro profiler

As you are probably now well aware, Go's pprof is extremely powerful and useful, if not exactly user-friendly.

If you love pprof already, or even if you find it arduous and confusing, you may love Nitro profiler twice as much. Coming from Steve Francia of spf13, Nitro profiler allows you to produce even cleaner analyses of your application and its functions and steps, as well as providing more usable a/b tests of alternate functions.



Read more about Nitro profiler at <http://spf13.com/project/nitro>.

You can get it via github.com/spf13/nitro.

As with pprof, Nitro automatically injects flags into your application, and you'll see them in the results themselves.

Unlike pprof, your application does not need to be compiled to get profile analysis from it. Instead, you can simply append `-stepAnalysis` to the `go run` command.

Heka

Heka is a data pipeline tool that can be used to gather, analyze, and distribute raw data. Available from Mozilla, Heka is more a standalone application rather than a library, but when it comes to acquiring, analyzing, and distributing data such as server logfiles across multiple servers, Heka can prove itself worthy.


Heka is also written in Go, so make sure to check out the source to see how Mozilla utilizes concurrency and Go in real-time data analysis.

[ You can visit the Heka home page at <http://heka-docs.readthedocs.org/en/latest/> and the Heka source page at <https://github.com/mozilla-services/heka>.]

GoFlow

Finally, there's GoFlow, a flow-based programming paradigm tool that lets you segment your application into distinct components, each capable of being bound to ports, channels, the network, or processes.

While not itself a performance tool, GoFlow might be an appropriate approach to extending concurrency for some applications.

[ Visit GoFlow at <https://github.com/trustmaster/goflow>.]

Memory preservation

At the time of this writing, Go 1.2.2's compiler utilizes a naive mark/sweep garbage collector, which assigns a reference rank to objects and clears them when they are no longer in use. This is noteworthy only to point out that it is widely considered a relatively poor garbage collection system.

So why does Go use it? As Go has evolved; language features and compiler speed have largely taken precedence over garbage collection. While it's a long-term development timeline for Go, for the time being, this is where we are. The tradeoff is a good one, though: as you well know by now, compiling Go code is light years faster than, say, compiling C or C++ code. Good enough for now is a fair description for the GC. But there are some things you can do to augment and experiment within the garbage collection system.

Garbage collection in Go

To get an idea of how the garbage collector is managing the stack at any time, take a look at the `runtime.MemProfileRecord` object, which keeps track of presently living objects in the active stack trace.

You can call the profile record when necessary and then utilize it against the following methods to get a few interesting pieces of data:

- `InUseBytes()`: This method has the bytes used presently as per the memory profile
- `InUseObjects()`: This method has the number of live objects in use
- `Stack()`: This method has the full stack trace

You can place the following code in a heavy loop in your application to get a peek at all of these:

```
var mem runtime.MemProfileRecord
obj := mem.InUseObjects();
bytes := mem.InUseBytes();
stack := mem.Stack();
fmt.Println(i, obj, bytes)
```

Summary

We can now build some pretty high-performance applications and then utilize some of Go's built-in tools and third-party packages to seek out the most performance in a single instance application as well as across multiple, distributed systems.

In the next chapter, we're going to wrap everything together to design and build a concurrent server application that can work quickly and independently, and easily scale in performance and scope.

8

Concurrent Application Architecture

By now, we've designed small bits of concurrent programs, primarily in a single piece keeping concurrency largely isolated. What we haven't done yet is tie everything together to build something a little more robust, complex, and more daunting to manage from an administrator's perspective.

Simple chat applications and web servers are fine and dandy. However, you will eventually need more complexity and require external software to meet all of the more advanced requirements.

In this case, we'll build something that's satisfied by a few dissonant services: a file manager with revision control that supplies web and shell access. Services such as Dropbox and Google Drive allow users to keep and share files among peers. On the other hand, GitHub and its ilk allow for a similar platform but with the critical added benefit of revision control.

Many organizations face problems with the following sharing and distribution options:

- Limitations on repositories, storage, or number of files
- Potential inaccessibility if the services are down
- Security concerns, particularly for sensitive information

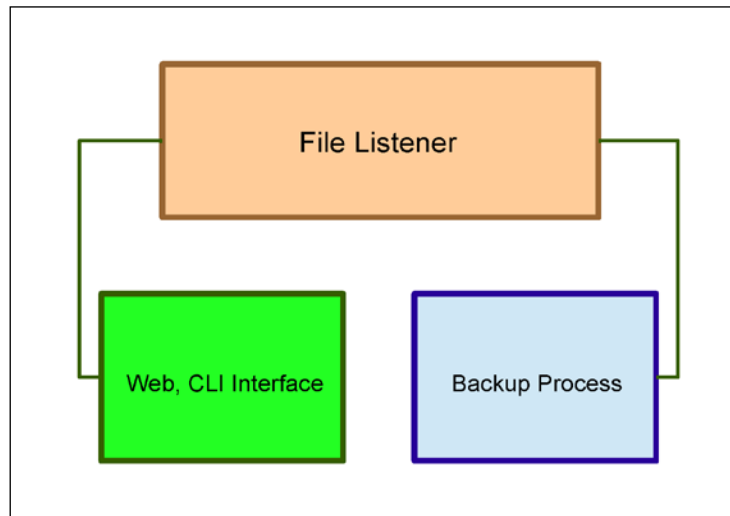
Simple sharing applications such as Dropbox and Google Drive are great at storing data without a large amount of revision control options. GitHub is an excellent collaborative revision control and distribution system, but comes with many costs and the mistakes by developers can lead to large and potentially serious security lapses.

We'll be combining the aims of version control (and the GitHub ideal) with Dropbox's / Google Drive's simplicity and openness. This type of application will be perfect as an intranet replacement—wholly isolated and accessible with custom authentication that doesn't necessarily rely on cloud services. The ability to keep it all in-house removes any potential for network security concerns and allows an administrator to design permanent backup solutions in a way that fits their organization.

File sharing within the organization will allow forking, backups, file locking, and revision control all from the command line but also through a simple web interface.

Designing our concurrent application

When designing a concurrent application, we will have three components running in separate processes. A file listener will be alerted to make changes to files in specified locations. A web-CLI interface will allow users to augment or modify files, and a backup process will be bound to the listener to provide automated copies of new file changes. With that in mind, these three processes will look a bit like what is shown in the following diagram:



Our file listener process will do the following three things:

- Keep an eye on any file changes
- Broadcast to our web/CLI servers and the backup process
- Maintain the state of any given file in our database / data store

The backup process will accept any broadcasts from the file listener (#2) and create a backup file in an iterative design.

Our general server (web and CLI) will report details on individual files and allow versioning forward and backward with a customizable syntax. This part of the application will also have to broadcast back to the file listener when new files are committed or revisions are requested.

Identifying our requirements

The most critical step in our architectural design process is really zooming in on the required features, packages, and technologies that we'll need to implement. For our file management and revision control application, there are a few key points that will stand out:

- A web interface that allows file uploads, downloads, and revisions.
- A command-line interface that allows us to roll back changes and modify files directly.
- A filesystem listener that finds changes made to a shared location.
- A data store system that has strong Go tie-in and allows us to maintain information about files and users in a mostly consistent manner. This system will also maintain user records.
- A concurrent log system that maintains and cycles logs of changed files.

We're somewhat complicating things by allowing the following three different ways to interface with the overall application:

- Via the Web that requires a user and login. This also allows our users to access and modify files even if they happen to be somewhere not connected to the shared drive.
- Via the command line. This is archaic but also extremely valuable anytime a user is traversing a filesystem, particularly power users not in a GUI.
- Via the filesystem that changes itself. This is the shared drive mechanism wherein we assume that any user with access to this will be making valid modifications to any files.

To handle all of this, we can identify a few critical technologies as follows:

- A database or data store to manage revisions to our filesystem. When choosing between transactional, ACID-compliant SQL and fast document stores in NoSQL, the tradeoff is often performance versus consistency. However, since most of our locking mechanism will exist in the application, duplicating locks (even at the row level) will add a level of potential slowness and cruft that we don't need. So, we will utilize a NoSQL solution.
- This solution will need to play well with concurrency.
- We'll be using a web interface, one that brings in powerful and clean routing/muxing and plays well with Go's robust built-in templating system.
- A filesystem notification library that allows us to monitor changes to files as well as backing up revisions.

Any solutions we uncover or build to satisfy these requirements will need to be highly concurrent and non-blocking. We'll want to make sure that we do not allow simultaneous changes to files, including changes to our internal revisions themselves.

With all of this in mind, let's identify our pieces one-by-one and decide how they will play in our application.

We'll also present a few alternatives with options that can be swapped without compromising the functionality or core requirements. This will allow some flexibility in cases where platform or preference makes our primary option unpalatable. Any time we're designing an application, it's a good idea to know what else is out there in case the software (or terms of its use) change or it is no longer satisfactory to use at a future scale.

Let's start with our data store.

Using NoSQL as a data store in Go

One of the biggest concessions with using NoSQL is, obviously, the lack of standardization when it comes to CRUD operations (create, read, update, and delete). SQL has been standardized since 1986 and is pretty airtight across a number of databases – from MySQL to SQL Server and from Microsoft and Oracle all the way down to PostgreSQL.



You can read more about NoSQL and various NoSQL platforms at <http://nosql-database.org/>.

Martin Fowler has also written a popular introduction to the concept and some use cases in his book *NoSQL Distilled* at <http://martinfowler.com/books/nosql.html>.

Depending on the NoSQL platform, you can also lose ACID compliance and durability. This means that your data is not 100 percent secure – there can be transactional loss if a server crashes, if reads happen on outdated or non-existent data, and so on. The latter of which is known as a dirty read.

This is all noteworthy as it applies to our application and with concurrency specifically because we've talked about one of those big potential third-party bottlenecks in the previous chapters.

For our file-sharing application in Go, we will utilize NoSQL to store metadata about files as well as the users that modify/interact with those files.

We have quite a few options when it comes to a NoSQL data store to use here, and almost all of the big ones have a library or interface in Go. While we're going to go with Couchbase here, we'll briefly talk about some of the other big players in the game as well as the merits of each.

The code snippets in the following sections should also give you some idea of how to switch out Couchbase for any of the others without too much angst. While we don't go deeply into any of them, the code for maintaining the file and modifying information will be as generic as possible to ensure easy exchange.

MongoDB

MongoDB is one of the most popular NoSQL platforms available. Written in 2009, it's also one of the most mature platforms, but comes with a number of tradeoffs that have pushed it somewhat out of favor in the recent years.

Even so, Mongo does what it does in a reliable fashion and with a great deal of speed. Utilizing indices, as is the case with most databases and data stores, improves query speed on reads greatly.

Mongo also allows for some very granular control of guarantees as they apply to reads, writes, and consistency. You can think of this as a very vague analog to any language and/or engine that supports syntactical dirty reads.

Most importantly, Mongo supports concurrency easily within Go and is implicitly designed to work in distributed systems.



The biggest Go interface for Mongo is mgo, which is available at:
<http://godoc.org/labix.org/v2/mgo>.

Should you wish to experiment with Mongo in Go, it's a relatively straightforward process to take your data store record and inject it into a custom struct. The following is a quick and dirty example:

```
import
(
    "labix.org/v2/mgo"
    "labix.org/v2/mgo/bson"
)

type User struct {
    name string
}

func main() {
    servers, err := mgo.Dial("localhost")
    defer servers.Close()
    data := servers.DB("test").C("users")
    result := User{}
    err = c.Find(bson.M{"name": "John"}).One(&result)
}
```

One downside to Mongo compared to other NoSQL solutions is that it does not come with any GUI by default. This means we either need to tie in another application or web service, or stick to the command line to manage its data store. For many applications, this isn't a big deal, but we want to keep this project as compartmentalized and provincial as possible to limit points of failure.

Mongo has also gotten a bit of a bad rap as it pertains to fault tolerance and data loss, but this is equally true of many NoSQL solutions. In addition, it's in many ways a feature of a fast data store—so often catastrophe recovery comes at the expense of speed and performance.

It's also fair to say this is a generally overblown critique of Mongo and its peers. Can something bad happen with Mongo? Sure. Can it also happen with a managed Oracle-based system? Absolutely. Mitigating massive failures in this realm is more the responsibility of a systems administrator than the software itself, which can only provide the tools necessary to design such a contingency plan.

All that said, we'll want something with a quick and highly-available management interface, so Mongo is out for our requirements but could easily be plugged into this solution if those are less highly valued.

Redis

Redis is another key/value data store and, as of recently, took the number one spot in terms of total usage and popularity. In an ideal Redis world, an entire dataset is held in memory. Given the size of many datasets, this isn't always possible; however, coupled with Redis' ability to eschew durability, this can result in some very high performance results when used in concurrent applications.

Another useful feature of Redis is the fact that it can inherently hold different data structures. While you can make abstractions of such data by unmarshalling JSON objects/arrays in Mongo (and other data stores), Redis can handle sets, strings, arrays, and hashes.

There are two major accepted libraries for Redis in Go:

- **Radix:** This is a minimalist client that's barebones, quick, and dirty. To install Radix, run the following command:
`go get github.com/fzzy/radix/redis`
- **Redigo:** This more robust and a bit more complex, but provides a lot of the more intricate functionality that we'll probably not need for this project. To install Redigo, run the following command:
`go get github.com/garyburd/redigo/redis`

We'll now see a quick example of getting a user's name from the data store of Users in Redis using Redigo:

```
package main

import (
    "fmt"
    "github.com/garyburd/redigo/redis"
)
```

```
func main() {

    connection, _ := dial()
    defer connection.Close()

    data, err := redis.Values(connection.Do("SORT", "Users", "BY",
"User:*->name",
    "GET", "User:*->name"))

    if (err) {
        fmt.Println("Error getting values", err)
    }

    for i:= range data {
        var Uname string
        data,err := redis.Scan(data, &Uname)
        if (err) {
            fmt.Println("Error getting value",err)
        }else {
            fmt.Println("Name Uname")
        }
    }
}
```

Looking over this, you might note some non programmatic access syntax, such as the following:

```
data, err := redis.Values(connection.Do("SORT", "Users", "BY",
"User:*->name",
    "GET", "User:*->name"))
```

This is indeed one of the reasons why Redis in Go will not be our choice for this project—both libraries here provide an almost API-level access to certain features with some more detailed built-ins for direct interaction. The `Do` command passes straight queries directly to Redis, which is fine if you need to use the library, but a somewhat inelegant solution across the board.

Both the libraries play very nicely with the concurrent features of Go, and you'll have no problem making non-blocking networked calls to Redis through either of them.

It's worth noting that Redis only supports an experimental build for Windows, so this is mostly for use on *nix platforms. The port that does exist comes from Microsoft and can be found at <https://github.com/Microsoft/redis>.

Tiedot

If you've worked a lot with NoSQL, then the preceding engines all likely seemed very familiar to you. Redis, Couch, Mongo, and so on are all virtual stalwarts in what is a relatively young technology.

Tiedot, on the other hand, probably isn't as familiar. We're including it here only because the document store itself is written in Go directly. Document manipulation is handled primarily through a web interface, and it's a JSON document store like several other NoSQL solutions.

As document access and handling is governed via HTTP, there's a somewhat counterintuitive workflow, shown as follows:

[Your Go Application] → Web Interface → [Go Application—Tiedot]

As that introduces a potential spot for latency or failure, this keeps from being an ideal solution for our application here. Keep in mind that this is also a feature of a few of the other solutions mentioned earlier, but since Tiedot is written in Go, it would be significantly easier to connect to it and read/modify data using a package. While this book was being written, this did not exist.

Unlike other HTTP- or REST-focused alternatives such as CouchDB, Tiedot relies on URL endpoints to dictate actions, not HTTP methods.

You can see in the following code how we might handle something like this through standard libraries:

```
package main

import (
    "fmt"
    "json"
    "http"
)

type Collection struct {
    Name string
}
```

This, simply, is a data structure for any record you wish to bring into your Go application via data selects, queries, and so on. You saw this in our previous usage of SQL servers themselves, and this is not any different:

```
func main() {  
  
    Col := Collection{  
        Name: ''  
    }  
  
    data, err := http.Get("http://localhost:8080/all")  
    if (err != nil) {  
        fmt.Println("Error accessing tiedot")  
    }  
    collections, _ = json.Unmarshal(data, &Col)  
}
```

While not as robust, powerful, or scalable as many of its peers, Tiedot is certainly worth playing with or, better yet, contributing to.



You can find Tiedot at <https://github.com/HouzuoGuo/tiedot>.

CouchDB

CouchDB from Apache Incubator is another one of the big boys in NoSQL big data. As a JSON document store, CouchDB offers a great deal of flexibility when it comes to your data store approach.

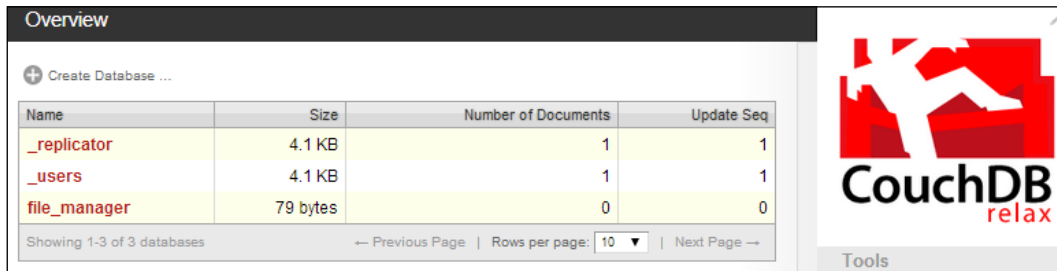
CouchDB supports ACID semantics and can do so concurrently, which provides a great deal of performance benefit if one is bound to those properties. In our application, that reliance on ACID consistency is somewhat flexible. By design, it will be failure tolerant and recoverable, but for many, even the possibility of data loss with recoverability is still considered catastrophic.

Interfacing with CouchDB happens via HTTP, which means there is no need for a direct implementation or Go SQL database hook to use it. Interestingly, CouchDB uses HTTP header syntax to manipulate data, as follows:

- **GET:** This represents read operations
- **PUT:** This represents creation operations
- **DELETE:** This represents deletion and update operations

These are, of course, what the header methods were initially intended in HTTP 1.1, but so much of the Web has focused on GET/POST that these tend to get lost in the fray.

Couch also comes with a convenient web interface for management. When CouchDB is running, you're able to access this at `http://localhost:5984/_utils/`, as shown in the following screenshot:



That said, there are a few wrappers that provide a level of abstraction for some of the more complicated and advanced features.

Cassandra

Cassandra, another Apache Foundation project, isn't technically a NoSQL solution but a clustered (or cluster-able) database management platform.

Like many NoSQL applications, there is a limitation in the traditional query methods in Cassandra, for example, subqueries and joins are generally not supported.

We're mentioning it here primarily because of its focus on distributed computing as well as the ability to programmatically tune whether data consistency or performance is more important. Much of that is equally expressed in our solution, Couchbase, but Cassandra has a deeper focus on distributed data stores.

Cassandra does, however, support a subset of SQL that will make it far more familiar to developers who have dabbled in MySQL, PostgreSQL, or the ilk. Cassandra's built-in handling of highly concurrent integrations makes it in many ways ideal for Go, although it is an overkill for this project.

The most noteworthy library to interface with Cassandra is `gocql`, which focuses on speed and a clean connection to the Cassandra connection. Should you choose to use Cassandra in lieu of Couchbase (or other NoSQL), you'll find a lot of the methods that can be simply replaced.

The following is an example of connecting to a cluster and writing a simple query:

```
package main

import (
    "github.com/gocql/gocql"
    "log"
)

func main() {

    cass := gocql.NewCluster("127.0.0.1")
    cass.Keyspace = "filemaster"
    cass.Consistency = gocql.LocalQuorum

    session, _ := cass.CreateSession()
    defer session.Close()

    var fileTime int;

    if err := session.Query(`SELECT file_modified_time FROM filemaster
WHERE filename = ? LIMIT 1`,
    "test.txt").Consistency(gocql.One).Scan(&fileTime); err != nil {
        log.Fatal(err)
    }
    fmt.Println("Last modified",fileTime)
}
```

Cassandra may be an ideal solution if you plan on rapidly scaling this application, distributing it widely, or are far more comfortable with SQL than data store / JSON access.

For our purposes here, SQL is not a requirement and we value speed over anything else, including durability.

Couchbase

Couchbase is a relative newcomer in the field, but it was built by people from both CouchDB and memcached. Written in Erlang, it shares many of the same focuses on concurrency, speed, and non-blocking behavior that we've come to expect from a great deal of our Go applications.

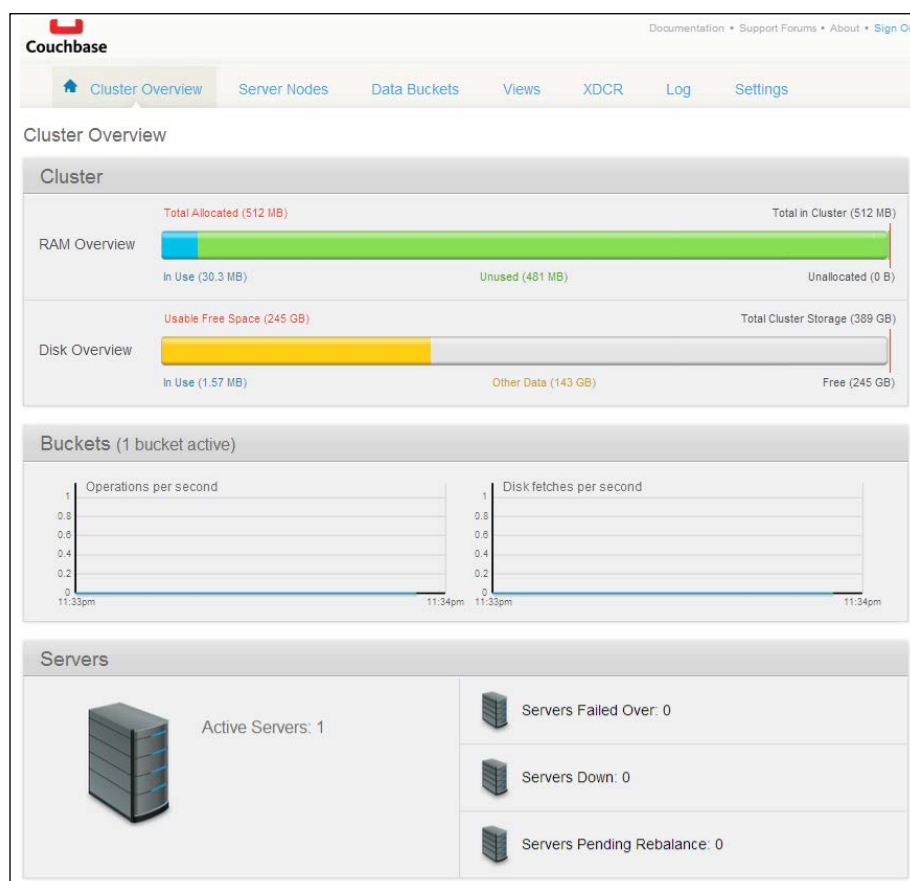
Couchbase also supports a lot of the other features we've discussed in the previous chapters, including easy distribution-based installations, tuneable ACID compliance, and low-resource consumption.

One caveat on Couchbase is it doesn't run well (or at all) on some lower-resourced machines or VMs. Indeed, 64-bit installations require an absolute minimum of 4 GB of memory and four cores, so forget about launching this on tiny, small, or even medium-grade instances or older hardware.

While most NoSQL solutions presented here (or elsewhere) offer performance benefits over their SQL counterparts in general, Couchbase has done very well against its peers in the NoSQL realm itself.

Couchbase, such as CouchDB, comes with a web-based graphical interface that simplifies the process of both setup and maintenance. Among the advanced features that you'll have available to you in the setup include your base bucket storage engine (Couchbase or memcached), your automated backup process (replicas), and the level of read-write concurrency.

In addition to configuration and management tools, it also presents some real-time monitoring in the web dashboard as shown in the following screenshot:



While not a replacement for full-scale server management (what happens when this server goes down and you have no insight), it's incredibly helpful to know exactly where your resources are going without needing a command-line method or an external tool.

The vernacular in Couchbase varies slightly, as it tends to in many of these solutions. The nascent desire to slightly separate NoSQL from stodgy old SQL solutions will pop its head from time to time.

With Couchbase, a database is a data bucket and records are documents. However, views, an old transactional SQL standby, bring a bit of familiarity to the table. The big takeaway here is views allow you to create more complex queries using simple JavaScript, in some cases, replicating otherwise difficult features such as joins, unions, and pagination.

Each view created in Couchbase becomes an HTTP access point. So a view that you name `select_all_files` will be accessible via a URL such as `http://localhost:8092/file_manager/_design/select_all_files/_view/Select%20All%20Files?connection_timeout=60000&limit=10&skip=0`.

The most noteworthy Couchbase interface library is Go Couchbase, which, if nothing else, might save you from some of the redundancy of making HTTP calls in your code to access CouchDB.



Go Couchbase can be found at <https://github.com/couchbaselabs/go-couchbase>.

Go Couchbase makes interfacing with Couchbase through a Go abstraction simple and powerful. The following code connects and grabs information about the various data pools in a lean way that feels native:

```
package main

import (
    "fmt"
    "github.com/couchbaselabs/go-couchbase"
)

func main() {

    conn, err := couchbase.Connect("http://localhost:8091")
    if err != nil {
        fmt.Println("Error:", err)
    }
}
```

```

for _, pn := range conn.Info.Pools {
    fmt.Printf("Found pool:  %s -> %s\n", pn.Name, pn.URI)
}
}

```

Setting up our data store

After installing Couchbase, you can access its administration panel by default at localhost and port 8091.

You'll be given an opportunity to set up an administrator, other IPs to connect (if you're joining a cluster), and general data store design.

After that, you'll need to set up a bucket, which is what we'll use to store all information about individual files. Here is what the interface for the bucket setup looks like:

Create Bucket

Bucket Settings

Bucket Name:

Bucket Type: ☒ Couchbase ☐ Memcached

Memory Size

Per Node RAM Quota: MB

Cluster quota (512 MB)

Other Buckets (0 B) This Bucket (256 MB) Free (256 MB)

Total bucket size = 256 MB (256 MB x 1 node)

Access Control

☒ Standard port (TCP port 11211. Needs SASL auth.)

Enter password:

☐ Dedicated port (supports ASCII protocol and is auth-less)

Protocol Port:

Replicas

☐ Enable

☐ Index replicas

Disk Read-Write Concurrency

Number of suggested reader/writer workers: (Min = 2, Max = 8)

Auto-Compaction

The Auto-Compaction daemon compacts databases and their respective view indexes when all the condition parameters are satisfied.

☐ Override the default autocompaction settings?

Flush

☐ Enable

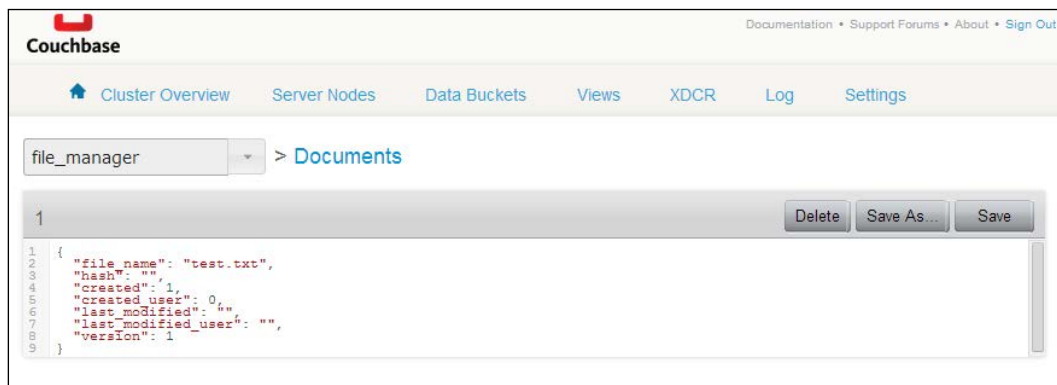
Cancel Create

In our example, we're working on a single machine, so replicas (also known as replication in database vernacular) are not supported. We've named it `file_manager`, but this can obviously be called anything that makes sense.

We're also keeping our data usage pretty low – there's no need for much more than 256 MB of memory when we're storing file operations and logging older ones. In other words, we're not necessarily concerned with keeping the modification history of `test.txt` in memory forever.

We'll also stick with Couchbase for a storage engine equivalent, although you can flip back and forth with memcache(d) without much noticeable change.

Let's start by creating a seed document: one we'll delete later, but that will represent the schema of our data store. We can create this document with an arbitrary JSON structured object, as shown in the following screenshot:



Since everything stored in this data store should be valid JSON, we can mix and match strings, integers, booleans, arrays, and objects. This affords us some flexibility in what data we're using. The following is an example document:

```
{
  "file_name": "test.txt",
  "hash": "",
  "created": 1,
  "created_user": 0,
  "last_modified": "",
  "last_modified_user": "",
  "revisions": [],
  "version": 1
}
```

Monitoring filesystem changes

When it came to NoSQL options, we had a vast variety of solutions at our disposal. This is not the case when it comes to applications that monitor filesystem changes. While Linux flavors have a fairly good built-in solution in inotify, this does restrict the portability of the application.

So it's incredibly helpful that a cross-platform library for handling this exists in Chris Howey's fsnotify.

Fsnotify works on Linux, OSX, and Windows and allows us to detect when files in any given directory are created, deleted, modified, or renamed, which is more than enough for our purposes.

Implementing fsnotify couldn't be easier, either. Best of all it's all non-blocking, so if we throw the listener behind a goroutine, we can have this run as part of the primary server application code.

The following code shows a simple directory listener:

```
package main

import (
    "github.com/howeyc/fsnotify"
    "fmt"
    "log"
)

func main() {

    scriptDone := make(chan bool)
    dirSpy, err := fsnotify.NewWatcher()
    if err != nil {
        log.Fatal(err)
    }

    go func() {
        for {
            select {
            case fileChange := <-dirSpy.Event:
                log.Println("Something happened to a file:",
                    fileChange)
            case err := <-dirSpy.Error:
                log.Println("Error with fsnotify:", err)
            }
        }
    }
```


```
    }  
  }()  
  
  err = dirSpy.Watch("/mnt/sharedir")  
  if err != nil {  
    fmt.Println(err)  
  }  
  
  <-scriptDone  
  
  dirSpy.Close()  
}
```

Managing logfiles

Like many basic features in a developer's toolbox, Go provides a fairly complete solution built-in for logging. It handles many of the basics, such as creating timestamp-marked log items and saving to disk or to console.

One thing the basic package misses out on is built-in formatting and log rotation, which are key requirements for our file manager application.

Remember that key requirements for our application include the ability to work seamlessly in our concurrent environment and be ready to scale to a distributed network if need be. This is where the fine **log4go** application comes in handy. Log4go allows logging to file, console, and memory and handles log rotation inherently.

[ Log4go can be found at <https://code.google.com/p/log4go/>.
To install Log4go, run the following command:
go get code.google.com/p/log4go]

Creating a logfile that handles warnings, notices, debug information, and critical errors is simple and appending log rotation to that is similarly simple, as shown in the following code:

```
package main  
  
import (  
    logger "code.google.com/p/log4go"  
)  
  
func main() {  
    logMech := make(logger.Logger);
```

```

logMech.AddFilter("stdout", logger.DEBUG,
    logger.NewConsoleLogWriter())

fileLog := logger.NewFileLogWriter("log_manager.log", false)
fileLog.SetFormat("[%D %T] [%L] (%S) %M")
fileLog.SetRotate(true)
fileLog.SetRotateSize(256)
fileLog.SetRotateLines(20)
fileLog.SetRotateDaily(true)
logMech.AddFilter("file", logger.FINE, fileLog)

logMech.Trace("Received message: %s)", "All is well")
logMech.Info("Message received: ", "debug!")
logMech.Error("Oh no!", "Something Broke")
}

```

Handling configuration files

When it comes to configuration files and parsing them, you have a lot of options, from simple to complicated.

We could, of course, simply store what we want in JSON, but that format is a little tricky to work directly for humans—it will require escaping characters and so on, which makes it vulnerable to errors.

Instead, we'll keep things simple by using a standard `ini` config file library in `gcfg`, which handles `gitconfig` files and traditional, old school `.ini` format, as shown in the following code snippet:

```

[revisions]
count = 2
revisionsuffix = .rev
lockfiles = false

[logs]
rotatelength = 86400

[alarms]
emails = sysadmin@example.com,ceo@example.com

```



You can find `gcfg` at <https://code.google.com/p/gcfg/>.

Essentially, this library takes the values of a config file and pushes them into a struct in Go. An example of how we'll do that is as follows:

```
package main

import (
    "fmt"
    "code.google.com/p/gcfg"
)

type Configuration struct {
    Revisions struct {
        Count int
        Revisionsuffix string
        Lockfiles bool
    }
    Logs struct {
        Rotatelength int
    }
    Alarms struct {
        Emails string
    }
}

func main() {
    configFile := Configuration{}
    err := gcfg.ReadFileInto(&configFile, "example.ini")
    if err != nil {
        fmt.Println("Error",err)
    }
    fmt.Println("Rotation duration:",configFile.Logs.Rotatelength)
}
```

Detecting file changes

Now we need to focus on our file listener. You may recall this is the part of the application that will accept client connections from our web server and our backup application and announce any changes to files.

The basic flow of this part is as follows:

1. Listen for changes to files in a goroutine.
2. Accept connections and add to the pool in a goroutine.
3. If any changes are detected, announce them to the entire pool.

All three happen concurrently, and the first and the third can happen without any connections in the pool, although we assume there will be a connection that is always on with both our web server and our backup application.

Another critical role the file listener will fulfill is analyzing the directory on first load and reconciling it with our data store in Couchbase. Since the Go Couchbase library handles the get, update, and add operations, we won't need any custom views. In the following code, we'll examine the file listener process and show how we listen on a folder for changes:

```
package main

import (
    "fmt"
    "github.com/howeyc/fsnotify"
    "net"
    "time"
    "io"
    "io/ioutil"
    "github.com/couchbaselabs/go-couchbase"
    "crypto/md5"
    "encoding/hex"
    "encoding/json"
    "strings"
)

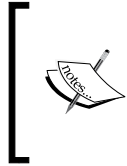
var listenFolder = "mnt/sharedir"

type Client struct {
    ID int
    Connection *net.Conn
}
```

Here, we've declared our shared folder as well as a connecting `Client` struct. In this application, `Client` is either a web listener or a backup listener, and we'll pass messages in one direction using the following JSON-encoded structure:

```
type File struct {
    Hash string "json:hash"
    Name string "json:file_name"
    Created int64 "json:created"
    CreatedUser int "json:created_user"
    LastModified int64 "json:last_modified"
    LastModifiedUser int "json:last_modified_user"
    Revisions int "json:revisions"
    Version int "json:version"
}
```


If this looks familiar, it could be because it's also the example document format we set up initially.



If you're not familiar with the syntactical sugar expressed earlier, these are known as struct tags. A tag is just a piece of additional metadata that can be applied to a struct field for key/value lookups via the reflect package. In this case, they're used to map our struct fields to JSON fields.

Let's first look at our overall `Message` struct:

```
type Message struct {
    Hash string "json:hash"
    Action string "json:action"
    Location string "json:location"
    Name string "json:name"
    Version int "json:version"
}
```

We compartmentalize our file into a message, which alerts our other two processes of changes:

```
func generateHash(name string) string {

    hash := md5.New()
    io.WriteString(hash, name)
    hashString := hex.EncodeToString(hash.Sum(nil))

    return hashString
}
```

This is a somewhat unreliable method to generate a hash reference to a file and will fail if a filename changes. However, it allows us to keep track of files that are created, deleted, or modified.

Sending changes to clients

Here is the broadcast message that goes to all existing connections. We pass along our JSON-encoded `Message` struct with the current version, the current location, and the hash for reference. Our other servers will then react accordingly:

```
func alertServers(hash string, name string, action string, location
string, version int) {
```

```

msg :=
    Message{Hash:hash,Action:action,Location:location,Name:name,
        Version:version}
msgJSON,_ := json.Marshal(msg)

fmt.Println(string(msgJSON))

for i := range Clients {
    fmt.Println("Sending to clients")
    fmt.Fprintln(*Clients[i].Connection,string(msgJSON))
}
}

```

Our backup server will create a copy of that file with the . [VERSION] extension in the backup folder.

Our web server will simply alert the user via our web interface that the file has changed:

```

func startServer(listener net.Listener) {
    for {
        conn,err := listener.Accept()
        if err != nil {

        }
        currentClient := Client{ ID: 1, Connection: &conn}
        Clients = append(Clients,currentClient)
        for i:= range Clients {
            fmt.Println("Client",Clients[i].ID)
        }
    }
}

```

Does this code look familiar? We've taken almost our exact chat server Client handler and brought it over here nearly intact:

```

func removeFile(name string, bucket *couchbase.Bucket) {
    bucket.Delete(generateHash(name))
}

```

The removeFile function does one thing only and that's removing the file from our Couchbase data store. As it's reactive, we don't need to do anything on the file-server side because the file is already deleted. Also, there's no need to delete any backups, as this allows us to recover. Next, let's look at our function that updates an existing file:

```
func updateExistingFile(name string, bucket *couchbase.Bucket) int {
    fmt.Println(name, "updated")
    hashString := generateHash(name)

    thisFile := Files[hashString]
    thisFile.Hash = hashString
    thisFile.Name = name
    thisFile.Version = thisFile.Version + 1
    thisFile.LastModified = time.Now().Unix()
    Files[hashString] = thisFile
    bucket.Set(hashString, 0, Files[hashString])
    return thisFile.Version
}
```

This function essentially overwrites any values in Couchbase with new ones, copying an existing File struct and changing the LastModified date:

```
func evalFile(event *fsnotify.FileEvent, bucket *couchbase.Bucket) {
    fmt.Println(event.Name, "changed")
    create := event.IsCreate()
    fileComponents := strings.Split(event.Name, "\\")
    fileComponentSize := len(fileComponents)
    trueFileName := fileComponents[fileComponentSize-1]
    hashString := generateHash(trueFileName)

    if create == true {
        updateFile(trueFileName, bucket)
        alertServers(hashString, event.Name, "CREATE", event.Name, 0)
    }
    delete := event.IsDelete()
    if delete == true {
        removeFile(trueFileName, bucket)
        alertServers(hashString, event.Name, "DELETE", event.Name, 0)
    }
    modify := event.IsModify()
    if modify == true {
        newVersion := updateExistingFile(trueFileName, bucket)
        fmt.Println(newVersion)
        alertServers(hashString, trueFileName, "MODIFY", event.Name,
            newVersion)
    }
    rename := event.IsRename()
    if rename == true {

    }
}
```

Here, we react to any changes to the filesystem in our watched directory. We aren't reacting to renames, but you can handle those as well. Here's how we'd approach the general `updateFile` function:

```
func updateFile(name string, bucket *couchbase.Bucket) {
    thisFile := File{}
    hashString := generateHash(name)

    thisFile.Hash = hashString
    thisFile.Name = name
    thisFile.Created = time.Now().Unix()
    thisFile.CreatedUser = 0
    thisFile.LastModified = time.Now().Unix()
    thisFile.LastModifiedUser = 0
    thisFile.Revisions = 0
    thisFile.Version = 1

    Files[hashString] = thisFile

    checkFile := File{}
    err := bucket.Get(hashString, &checkFile)
    if err != nil {
        fmt.Println("New File Added", name)
        bucket.Set(hashString, 0, thisFile)
    }
}
```

Checking records against Couchbase

When it comes to checking for existing records against Couchbase, we check whether a hash exists in our Couchbase bucket. If it doesn't, we create it. If it does, we do nothing. To handle shutdowns more robustly, we should also ingest existing records into our application. The code for doing this is as follows:

```
var Clients []Client
var Files map[string] File

func main() {
    Files = make(map[string]File)
    endScript := make(chan bool)

    couchbaseClient, err := couchbase.Connect("http://localhost:8091/")
    if err != nil {
```

```
        fmt.Println("Error connecting to Couchbase", err)
    }
    pool, err := couchbaseClient.GetPool("default")
    if err != nil {
        fmt.Println("Error getting pool",err)
    }
    bucket, err := pool.GetBucket("file_manager")
    if err != nil {
        fmt.Println("Error getting bucket",err)
    }

    files, _ := ioutil.ReadDir(listenFolder)
    for _, file := range files {
        updateFile(file.Name(),bucket)
    }

    dirSpy, err := fsnotify.NewWatcher()
    defer dirSpy.Close()

    listener, err := net.Listen("tcp", ":9000")
    if err != nil {
        fmt.Println ("Could not start server!",err)
    }

    go func() {
        for {
            select {
            case ev := <-dirSpy.Event:
                evalFile(ev,bucket)
            case err := <-dirSpy.Error:
                fmt.Println("error:", err)
            }
        }
    }()
    err = dirSpy.Watch(listenFolder)
    startServer(listener)

<-endScript
}
```

Finally, `main()` handles setting up our connections and goroutines, including a file watcher, our TCP server, and connecting to Couchbase.

Now, let's look at another step in the whole process where we will automatically create backups of our modified files.

Backing up our files

Since we're sending our commands on the wire, so to speak, our backup process needs to listen on that wire and respond with any changes. Given that modifications will be sent via localhost, we should have minimal latency on both the network and the file side.

We'll also return some information as to what happened with the file, although at this point we're not doing much with that information. The code for this is as follows:

```
package main

import (
    "fmt"
    "net"
    "io"
    "os"
    "strconv"
    "encoding/json"
)

var backupFolder = "mnt/backup/"
```

Note that we have a separate folder for backups, in this case, on a Windows machine. If we happen to accidentally use the same directory, we run the risk of infinitely duplicating and backing up files. In the following code snippet, we'll look at the Message struct itself and the backup function, the core of this part of the application:

```
type Message struct {
    Hash string "json:hash"
    Action string "json:action"
    Location string "json:location"
    Name string "json:name"
    Version int "json:version"
}

func backup (location string, name string, version int) {

    newFileName := backupFolder + name + "." +
        strconv.FormatInt(int64(version), 10)
    fmt.Println(newFileName)
    org, _ := os.Open(location)
    defer org.Close()
    cpy, _ := os.Create(newFileName)
    defer cpy.Close()
    io.Copy(cpy, org)
}
```

Here is our basic file operation. Go doesn't have a one-step copy function; instead you need to create a file and then copy the contents of another file into it with `io.Copy`:

```
func listen(conn net.Conn) {
    for {

        messBuff := make([]byte, 1024)
        n, err := conn.Read(messBuff)
        if err != nil {

        }

        resultMessage := Message{}
        json.Unmarshal(messBuff[:n], &resultMessage)

        if resultMessage.Action == "MODIFY" {
            fmt.Println("Back up file", resultMessage.Location)
            newVersion := resultMessage.Version + 1
            backup(resultMessage.Location, resultMessage.Name, newVersion)
        }

    }
}
```

This code is nearly verbatim for our chat client's `listen()` function, except that we take the contents of the streamed JSON data, unmarshal it, and convert it to a `Message{}` struct and then a `File{}` struct. Finally, let's look at the main function and TCP initialization:

```
func main() {
    endBackup := make(chan bool)
    conn, err := net.Dial("tcp", "127.0.0.1:9000")
    if err != nil {
        fmt.Println("Could not connect to File Listener!")
    }
    go listen(conn)

    <- endBackup
}
```

Designing our web interface

To interact with the filesystem, we'll want an interface that displays all of the current files with the version, last modified time, and alerts to changes, and allows drag-and-drop creation/replacement of files.

Getting a list of files will be simple, as we'll grab them directly from our `file_manager` Couchbase bucket. Changes will be sent through our file manager process via TCP, which will trigger an API call, illuminating changes to the file for our web user.

A few of the methods we've used here are duplicates of the ones we used in the backup process and could certainly benefit from some consolidation; still, the following is the code for the web server, which allows uploads and shows notifications for changes:

```
package main

import (
    "net"
    "net/http"
    "html/template"
    "log"
    "io"
    "os"
    "io/ioutil"
    "github.com/couchbaselabs/go-couchbase"
    "time"
    "fmt"
    "crypto/md5"
    "encoding/hex"
    "encoding/json"
)

type File struct {
    Hash string "json:hash"
    Name string "json:file_name"
    Created int64 "json:created"
    CreatedUser int "json:created_user"
    LastModified int64 "json:last_modified"
    LastModifiedUser int "json:last_modified_user"
    Revisions int "json:revisions"
    Version int "json:version"
}
```


This, for example, is the same `File` struct we use in both the file listener and the backup process:

```
type Page struct {
    Title string
    Files map[string] File
}
```

Our `Page` struct represents generic web data that gets converted into corresponding variables for our web page's template:

```
type ItemWrapper struct {

    Items []File
    CurrentTime int64
    PreviousTime int64

}

type Message struct {
    Hash string "json:hash"
    Action string "json:action"
    Location string "json:location"
    Name string "json:name"
    Version int "json:version"
}
```

The `ItemWrapper` struct is simply a JSON wrapper that will keep an array that's converted from our `Files` struct. This is essential to loop through the API's JSON on the client side. Our `Message` struct is a duplicate of the same type we saw in our file listener and backup processes. In the following code snippet, we'll dictate some general configuration variables and our hash generation function:

```
var listenFolder = "/wamp/www/shared/"
var Files map[string] File
var webTemplate = template.Must(template.ParseFiles("ch8_html.html"))
var fileChange chan File
var lastChecked int64

func generateHash(name string) string {

    hash := md5.New()
    io.WriteString(hash, name)
    hashString := hex.EncodeToString(hash.Sum(nil))

    return hashString
}
```

Our md5 hashing method is the same for this application as well. It's worth noting that we derive a `lastChecked` variable that is the Unix-style timestamp from each time we get a signal from our file listener. We use this to compare with file changes on the client side to know whether to alert the user on the Web. Let's now look at the `updateFile` function for the web interface:

```
func updateFile(name string, bucket *couchbase.Bucket) {
    thisFile := File{}
    hashString := generateHash(name)

    thisFile.Hash = hashString
    thisFile.Name = name
    thisFile.Created = time.Now().Unix()
    thisFile.CreatedUser = 0
    thisFile.LastModified = time.Now().Unix()
    thisFile.LastModifiedUser = 0
    thisFile.Revisions = 0
    thisFile.Version = 1

    Files[hashString] = thisFile

    checkFile := File{}
    err := bucket.Get(hashString, &checkFile)
    if err != nil {
        fmt.Println("New File Added", name)
        bucket.Set(hashString, 0, thisFile)
    } else {
        Files[hashString] = checkFile
    }
}
```

This is the same function as our backup process, except that instead of creating a duplicate file, we simply overwrite our internal `File` struct to allow it represent its updated `LastModified` value when the `/api` is next called. And as with our last example, let's check out the `listen()` function:

```
func listen(conn net.Conn) {
    for {

        messBuff := make([]byte, 1024)
        n, err := conn.Read(messBuff)
        if err != nil {
```

```
    }
    message := string(messBuff[:n])
    message = message[0:]

    resultMessage := Message{}
    json.Unmarshal(messBuff[:n], &resultMessage)

    updateHash := resultMessage.Hash
    tmp := Files[updateHash]
    tmp.LastModified = time.Now().Unix()
    Files[updateHash] = tmp
  }
}
```

Here is where our message is read, unmarshalled, and set to its hashed map's key. This will create a file if it doesn't exist or update our current one if it does. Next, we'll look at the `main()` function, which sets up our application and the web server:

```
func main() {
    lastChecked := time.Now().Unix()
    Files = make(map[string]File)
    fileChange = make(chan File)
    couchbaseClient, err := couchbase.Connect("http://localhost:8091/")
    if err != nil {
        fmt.Println("Error connecting to Couchbase", err)
    }
    pool, err := couchbaseClient.GetPool("default")
    if err != nil {
        fmt.Println("Error getting pool", err)
    }
    bucket, err := pool.GetBucket("file_manager")
    if err != nil {
        fmt.Println("Error getting bucket", err)
    }

    files, _ := ioutil.ReadDir(listenFolder)
    for _, file := range files {
        updateFile(file.Name(), bucket)
    }

    conn, err := net.Dial("tcp", "127.0.0.1:9000")
    if err != nil {
        fmt.Println("Could not connect to File Listener!")
    }
}
```

```

go listen(conn)

http.HandleFunc("/api", func(w http.ResponseWriter, r
    *http.Request) {
    apiOutput := ItemWrapper{}
    apiOutput.PreviousTime = lastChecked
    lastChecked = time.Now().Unix()
    apiOutput.CurrentTime = lastChecked

    for i:= range Files {
        apiOutput.Items = append(apiOutput.Items,Files[i])
    }
    output,_ := json.Marshal(apiOutput)
    fmt.Fprintln(w,string(output))

})
http.HandleFunc("/", func(w http.ResponseWriter, r
    *http.Request) {
    output := Page{Files:Files,Title:"File Manager"}
    tmp, _ := template.ParseFiles("ch8_html.html")
    tmp.Execute(w, output)
})
http.HandleFunc("/upload", func(w http.ResponseWriter, r
    *http.Request) {
    err := r.ParseMultipartForm(10000000)
    if err != nil {
        return
    }
    form := r.MultipartForm

    files := form.File["file"]
    for i, _ := range files {
        newFileName := listenFolder + files[i].Filename
        org,_:= files[i].Open()
        defer org.Close()
        cpy,_ := os.Create(newFileName)
        defer cpy.Close()
        io.Copy(cpy,org)
    }
})

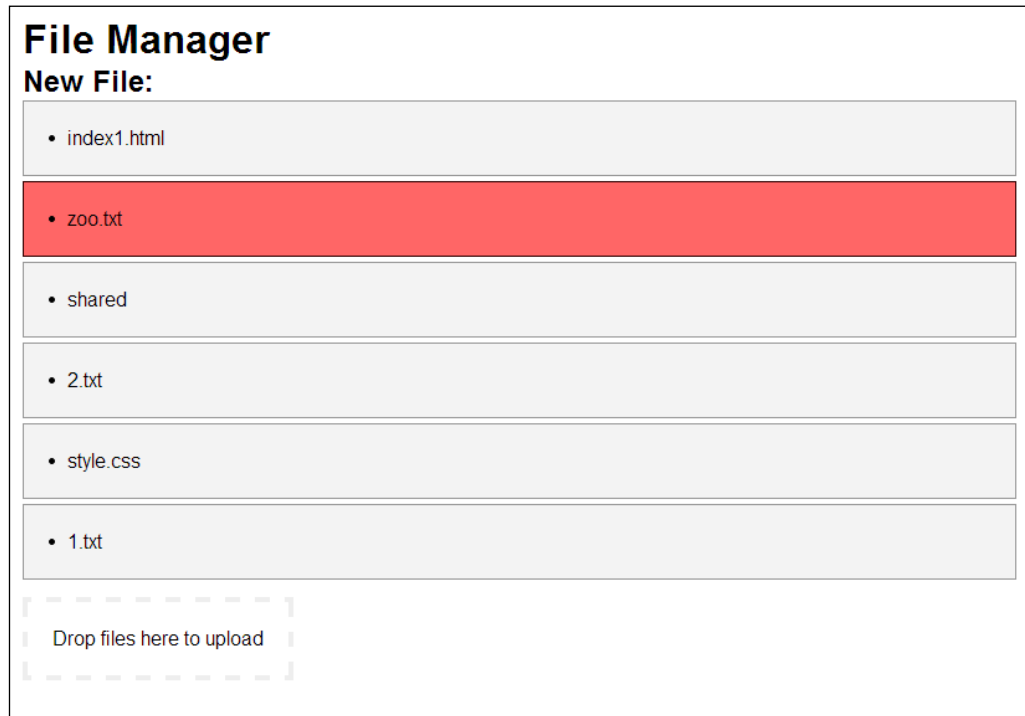
log.Fatal(http.ListenAndServe(":8080",nil))

}

```

In our web server component, `main()` takes control of setting up the connection to the file listener and Couchbase and creating a web server (with related routing).

If you upload a file by dragging it to the **Drop files here to upload** box, within a few seconds you'll see that the file is noted as changed in the web interface, as shown in the following screenshot:



We haven't included the code for the client side of the web interface; the key points, though, are retrieval via an API. We used a JavaScript library called `Dropzone.js` that allows a drag-and-drop upload, and jQuery for API access.

Reverting a file's history – command line

The final component we'd like to add to this application suite is a command-line file revision process. We can keep this one fairly simple, as we know where a file is located, where its backups are located, and how to replace the former with the latter. As with before, we have some global configuration variables and a replication of our `generateHash()` function:

```
var liveFolder = "/mnt/sharedir "
var backupFolder = "/mnt/backup

func generateHash(name string) string {

    hash := md5.New()
    io.WriteString(hash,name)
    hashString := hex.EncodeToString(hash.Sum(nil))

    return hashString
}

func main() {
    revision := flag.Int("r",0,"Number of versions back")
    fileName := flag.String("f","", "File Name")
    flag.Parse()

    if *fileName == "" {

        fmt.Println("Provide a file name to use!")
        os.Exit(0)
    }

    couchbaseClient, err := couchbase.Connect("http://localhost:8091/")
    if err != nil {
        fmt.Println("Error connecting to Couchbase", err)
    }
    pool, err := couchbaseClient.GetPool("default")
    if err != nil {
        fmt.Println("Error getting pool",err)
    }
    bucket, err := pool.GetBucket("file_manager")
    if err != nil {
        fmt.Println("Error getting bucket",err)
    }

    hashString := generateHash(*fileName)
    checkFile := File{}
    bucketerr := bucket.Get(hashString,&checkFile)
    if bucketerr != nil {
```

```
    }else {
        backupLocation := backupFolder + checkFile.Name + "." +
            strconv.FormatInt(int64(checkFile.Version-*revision),10)
        newLocation := liveFolder + checkFile.Name
        fmt.Println(backupLocation)
        org,_ := os.Open(backupLocation)
        defer org.Close()
        cpy,_ := os.Create(newLocation)
        defer cpy.Close()
        io.Copy(cpy,org)
        fmt.Println("Revision complete")
    }
}
```

This application accepts up to two parameters:

- -f: This denotes the filename
- -r: This denotes the number of versions to revert

Note that this itself creates a new version and thus a backup, so -2 would need to become -3, and then -6, and so on in order to continuously back up recursively.

As an example, if you wished to revert `example.txt` back three versions, you could use the following command:

```
fileversion -f example.txt -r -3
```

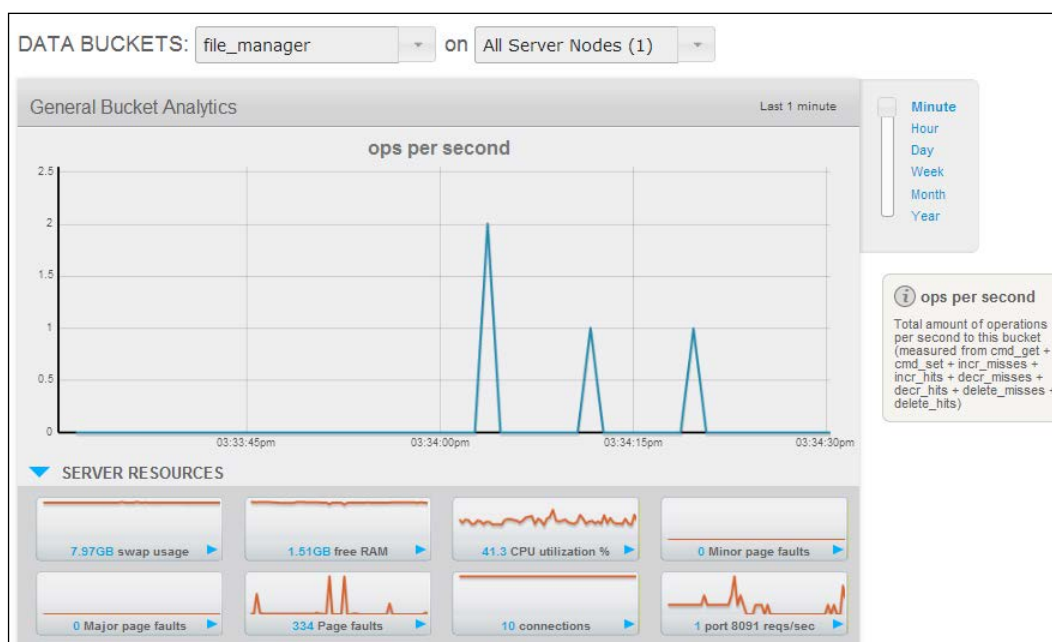
Using Go in daemons and as a service

A minor note on running something like this part of the application—you'll ideally wish to keep these applications as active, restartable services instead of standalone, manually executed background processes. Doing so will allow you to keep the application active and manage its life from external or server processes.

This sort of application suite would be best suited on a Linux box (or boxes) and managed with a daemon manager such as daemontools or Ubuntu's built-in Upstart service. The reason for this is that any long-term downtime can result in lost data and inconsistency. Even storing file data details in the memory (Couchbase and memcached) provides a vulnerability for lost data.

Checking the health of our server

Of the many ways to check general server health, we're in a good position here without having to build our own system, thanks in great part to Couchbase itself. If you visit the Couchbase web admin, under your cluster, server, and bucket views, clicking on any will present some real-time statistics, as shown in the following screenshot:



These areas are also available via REST if you wish to include them in the application to make your logging and error handling more comprehensive.

Summary

We now have a top to bottom application suite that is highly concurrent, ropes in several third-party libraries, and mitigates potential failures with logging and catastrophe recovery.

At this point, you should have no issue constructing a complex package of software with a focus on maintaining concurrency, reliability, and performance in Go. Our file monitoring application can be easily modified to do more, use alternative services, or scale to a robust, distributed environment.

In the next chapter, we'll take a closer look at testing our concurrency and throughput, explore the value of panic and recover, as well as dealing with logging vital information and errors in a safe, concurrent manner in Go.

9

Logging and Testing Concurrency in Go

At this stage, you should be fairly comfortable with concurrency in Go and should be able to implement basic goroutines and concurrent mechanisms with ease.

We have also dabbled in some distributed concurrency patterns that are managed not only through the application itself, but also through third-party data stores for networked applications that operate concurrently in congress.

Earlier in this book, we examined some preliminary and basic testing and logging. We looked at the simpler implementations of Go's internal test tool, performed some race condition testing using the race tool, and performed some rudimentary load and performance testing.

However, there's much more to be looked at here, particularly as it relates to the potential black hole of concurrent code – we've seen unexpected behavior among code that runs in goroutines and is non-blocking.

In this chapter, we'll further investigate load and performance testing, look at unit testing in Go, and experiment with more advanced tests and debugging. We'll also look at best practices for logging and reporting, as well as take a closer look at panicking and recovering.

Lastly, we'll want to see how all of these things can be applied not just to our standalone concurrent code, but also to distributed systems.

Along the way, we'll introduce a couple of frameworks for unit testing in a variety of different styles.

Handling errors and logging

Though we haven't specifically mentioned it, the idiomatic nature of error handling in Go makes debugging naturally easier by mandate.

One good practice for any large-scale function inside Go code is to return an error as a return value—for many smaller methods and functions, this is potentially burdensome and unnecessary. Still, it's a matter for consideration whenever we're building something that involves a lot of moving pieces.

For example, consider a simple `Add()` function:

```
func Add(x int, y int) int {  
    return x + y  
}
```

If we wish to follow the general rule of "always return an error value", we may be tempted to convert this function to the following code:

```
package main  
import  
(  
    "fmt"  
    "errors"  
    "reflect"  
)  
  
func Add(x int, y int) (int, error) {  
    var err error  
  
    xType := reflect.TypeOf(x).Kind()  
    yType := reflect.TypeOf(y).Kind()  
    if xType != reflect.Int || yType != reflect.Int {  
        fmt.Println(xType)  
        err = errors.New("Incorrect type for integer a or b!")  
    }  
    return x + y, err  
}  
  
func main() {  
  
    sum, err := Add("foo", 2)  
    if err != nil {  
        fmt.Println("Error", err)  
    }  
    fmt.Println(sum)  
}
```

You can see that we're (very poorly) reinventing the wheel. Go's internal compiler kills this long before we ever see it. So, we should focus on things that the compiler may not catch and that can cause unexpected behavior in our applications, particularly when it comes to channels and listeners.

The takeaway is to let Go handle the errors that the compiler would handle, unless you wish to handle the exceptions yourself, without causing the compiler specific grief. In the absence of true polymorphism, this is often cumbersome and requires the invocation of interfaces, as shown in the following code:

```
type Alpha struct {  
  
}  
  
type Numeric struct {  
  
}
```

You may recall that creating interfaces and structs allows us to route our function calls separately based on type. This is shown in the following code:

```
func (a Alpha) Add(x string, y string) (string, error) {  
    var err error  
    xType := reflect.TypeOf(x).Kind()  
    yType := reflect.TypeOf(y).Kind()  
    if xType != reflect.String || yType != reflect.String {  
        err = errors.New("Incorrect type for strings a or b!")  
    }  
    finalString := x + y  
    return finalString, err  
}  
  
func (n Numeric) Add(x int, y int) (int, error) {  
    var err error  
  
    xType := reflect.TypeOf(x).Kind()  
    yType := reflect.TypeOf(y).Kind()  
    if xType != reflect.Int || yType != reflect.Int {  
        err = errors.New("Incorrect type for integer a or b!")  
    }  
    return x + y, err  
}
```

```
func main() {
    n1 := Numeric{}
    a1 := Alpha{}
    z,err := n1.Add(5,2)
    if err != nil {
        log.Println("Error",err)
    }
    log.Println(z)

    y,err := a1.Add("super","lative")
    if err != nil {
        log.Println("Error",err)
    }
    log.Println(y)
}
```

This still reports what will eventually be caught by the compiler, but also handles some form of error on what the compiler cannot see: external input. We're routing our `Add()` function through an interface, which provides some additional standardization by directing the struct's parameters and methods more explicitly.

If, for example, we take user input for our values and need to evaluate the type of that input, we may wish to report an error in this way as the compiler will never know that our code can accept the wrong type.

Breaking out goroutine logs

One way of handling messaging and logging that keeps a focus on concurrency and isolation is to shackle our goroutine with its own logger that will keep everything separate from the other goroutines.

At this point, we should note that this may not scale—that is, it may at some point become expensive to create thousands or tens of thousands of goroutines that have their own loggers, but at a minimal size, this is totally doable and manageable.

To do this logging individually, we'll want to tie a `Logger` instance to each goroutine, as shown in the following code:

```
package main

import (
    "log"
```

```

    "os"
    "strconv"
)

const totalGoroutines = 5

type Worker struct {
    wLog *log.Logger
    Name string
}

```

We'll create a generic `Worker` struct that will ironically do no work (at least not in this example) other than hold onto its own `Logger` object. The code is as follows:

```

func main() {
    done := make(chan bool)

    for i:=0; i< totalGoroutines; i++ {

        myWorker := Worker{}
        myWorker.Name = "Goroutine " + strconv.FormatInt(int64(i),10) + " "
        myWorker.wLog = log.New(os.Stderr, myWorker.Name, 1)
        go func(w *Worker) {

            w.wLog.Print("Hmm")

            done <- true
        }(&myWorker)
    }
}

```

Each goroutine is saddled with its own log routine through `Worker`. While we're spitting our output straight to the console, this is largely unnecessary. However, if we want to siphon each to its own logfile, we could do so by using the following code:

```

    log.Println("...")

    <- done
}

```

Using the LiteIDE for richer and easier debugging

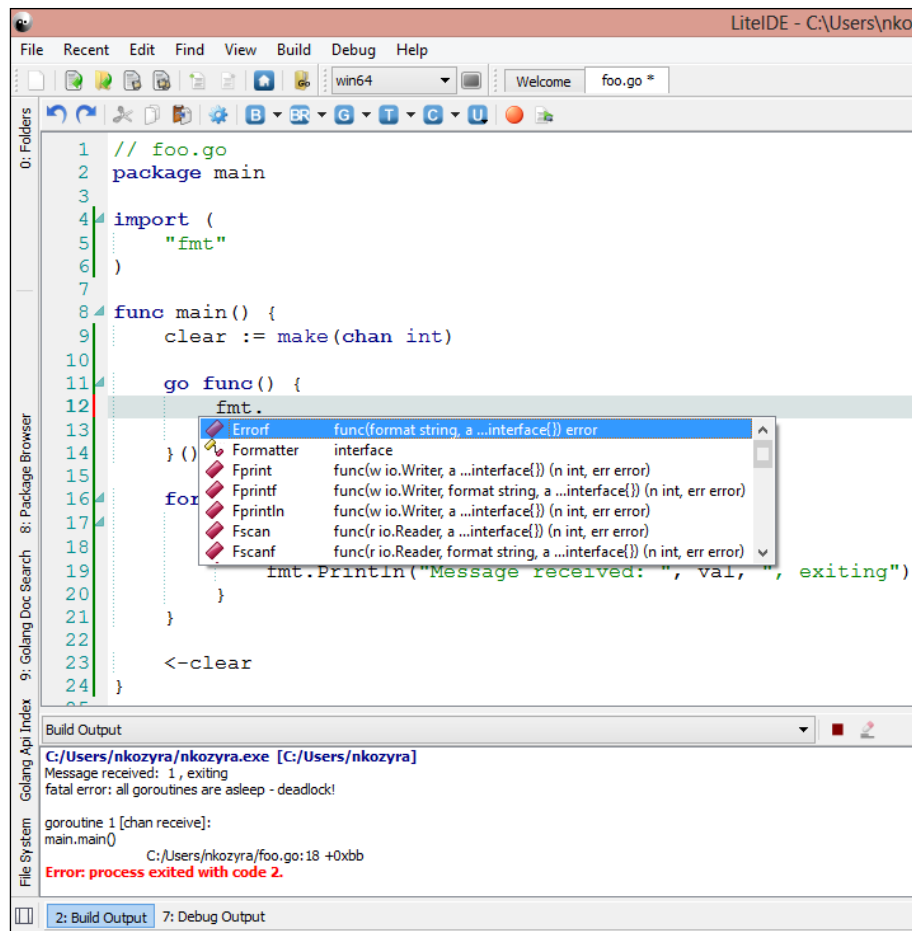
In the earlier chapters of this book, we briefly addressed IDEs and gave a few examples of IDEs that have a tight integration with Go.

As we're examining logging and debugging, there's one IDE we previously and specifically didn't mention before, primarily because it's intended for a very small selection of languages – namely, Go and Lua. However, if you end up working primarily or exclusively in Go, you'll find it absolutely essential, primarily as it relates to debugging, logging, and feedback capabilities.


LiteIDE is cross-platform and works well on OS X, Linux, and Windows. The number of debugging and testing benefits it presents in a GUI form are invaluable, particularly if you're already very comfortable with Go. That last part is important because developers often benefit most from "learning the hard way" before diving in with tools that simplify the programming process. It's almost always better to know how and why something works or doesn't work at the core before being presented with pretty icons, menus, and pop-up windows. Having said that, LiteIDE is a fantastic, free tool for the advanced Go programmer.

By formalizing a lot of the tools and error reporting from Go, we can easily plow through some of the more vexing debugging tasks by seeing them onscreen.

LiteIDE also brings context awareness, code completion, `go fmt`, and more into our workspace. You can imagine how an IDE tuned specifically for Go can help you keep your code clean and bug free. Refer to the following screenshot:



LiteIDE showing output and automatic code completion on Windows

[ LiteIDE for Linux, OS X, and Windows can be found at <https://code.google.com/p/liteide/>.]

Sending errors to screen

Throughout this book, we have usually handled soft errors, warnings, and general messages with the `fmt.Println` syntax by sending a message to the console.

While this is quick and easy for demonstration purposes, it's probably ideal to use the `log` package to handle these sorts of things. This is because we have more versatility, as `log` relates to where we want our messages to end up.

As for our purposes so far, the messages are ethereal. Switching out a simple `Println` statement to `Logger` is extremely simple.

We've been relaying messages before using the following line of code:

```
fmt.Println("Horrible error:", err)
```

You'll notice the change to `Logger` proves pretty similar:

```
myLogger.Println("Horrible error:", err)
```

This is especially useful for goroutines, as we can create either a global `Logger` interface that can be accessed anywhere or pass the logger's reference to individual goroutines and ensure our logging is handled concurrently.

One consideration for having a single logger for use across our entire application is the possibility that we may want to log individual processes separately for clarity in analysis. We'll talk a bit more about that later in this chapter.

To replicate passing messages to the command line, we can simply use the following line of code:

```
log.Print("Message")
```

With defaults to `stdout` as its `io.writer`—recall that we can set any `io.writer` as the log's destination.

However, we will also want to be able to log to file quickly and easily. After all, any application running in the background or as a daemon will need to have something a little more permanent.

Logging errors to file

There are a lot of ways to send an error to a logfile—we can, after all, handle this with built-in file operation OS calls. In fact, this is what many people do.

However, the `log` package offers some standardization and potential symbiosis between the command-line feedback and more permanent storage of errors, warnings, and general information.

The simplest way to do this is to open a file using the `os.OpenFile()` method (and not the `os.Open()` method) and pass that reference to our log instantiation as `io.Writer`.

Let's take a look at such functionality in the following example:

```
package main

import (
    "log"
    "os"
)

func main() {
    logFile, _ := os.OpenFile("/var/www/test.log", os.O_RDWR, 0755)

    log.SetOutput(logFile)
    log.Println("Sending an entry to log!")

    logFile.Close()
}
```

In our preceding goroutine package, we could assign each goroutine its own file and pass a file reference as an io Writer (we'll need to have write access to the destination folder). The code is as follows:

```
for i:=0; i< totalGoroutines; i++ {

    myWorker := Worker{}
    myWorker.Name = "Goroutine " + strconv.FormatInt(int64(i),10)
    + " "
    myWorker.FileName = "/var/www/"+strconv.FormatInt(int64(i),10)
    + ".log"
    tmpFile,_ := os.OpenFile(myWorker.FileName, os.O_CREATE,
        0755)
    myWorker.File = tmpFile
    myWorker.wLog = log.New(myWorker.File, myWorker.Name, 1)
    go func(w *Worker) {

        w.wLog.Print("Hmm")

        done <- true
    }(&myWorker)
}
```

Logging errors to memory

When we talk about logging errors to memory, we're really referring to a data store, although there's certainly no reason other than volatility and limited resources to reject logging to memory as a viable option.

While we'll look at a more direct way to handle networked logging through another package in the next section, let's delineate our various application errors in a concurrent, distributed system without a lot of hassle. The idea is to use shared memory (such as Memcached or a shared memory data store) to pass our log messages.

While these will technically still be logfiles (most data stores keep individual records or documents as JSON-encoded hard files), it has a distinctively different feel than traditional logging.

Going back to our old friend from the previous chapter – CouchDB – passing our logging messages to a central server can be done almost effortlessly, and it allows us to track not just individual machines, but their individual concurrent goroutines. The code is as follows:

```
package main

import (
    "github.com/couchbaselabs/go-couchbase"
    "io"
    "time"
    "fmt"
    "os"
    "net/http"
    "crypto/md5"
    "encoding/hex"
)

type LogItem struct {
    ServerID string "json:server_id"
    Goroutine int "json:goroutine"
    Timestamp time.Time "json:time"
    Message string "json:message"
    Page string "json:page"
}
```

This is what will eventually become our JSON document that will be sent to our Couchbase server. We'll use the `Page`, `Timestamp`, and `ServerID` as a combined, hashed key to allow multiple, concurrent requests to the same document against separate servers to be logged separately, as shown in the following code:

```
var currentGoroutine int

func (li LogItem) logRequest(bucket *couchbase.Bucket) {

    hash := md5.New()
    io.WriteString(hash, li.ServerID+li.Page+li.Timestamp.Format("Jan
        1, 2014 12:00am"))
    hashString := hex.EncodeToString(hash.Sum(nil))
    bucket.Set(hashString, 0, li)
    currentGoroutine = 0
}
```

When we reset `currentGoroutine` to 0, we use an intentional race condition to allow goroutines to report themselves by numeric ID while executing concurrently. This allows us to debug an application that appears to work correctly until it invokes some form of concurrent architecture. Since goroutines will be self-identified by an ID, it allows us to add more granular routing of our messages.

By designating a different log location by goroutine ID, timestamp, and serverID, any concurrency issues that arise can be quickly plucked from logfiles. This is done using the following code:

```
func main() {
    hostName, _ := os.Hostname()
    currentGoroutine = 0

    logClient, err := couchbase.Connect("http://localhost:8091/")
    if err != nil {
        fmt.Println("Error connecting to logging client", err)
    }
    logPool, err := logClient.GetPool("default")
    if err != nil {
        fmt.Println("Error getting pool", err)
    }
    logBucket, err := logPool.GetBucket("logs")
    if err != nil {
        fmt.Println("Error getting bucket", err)
    }
}
```

```
http.HandleFunc("/", func(w http.ResponseWriter, r
    *http.Request) {
    request := LogItem{}
    request.Goroutine = currentGoroutine
    request.ServerID = hostName
    request.Timestamp = time.Now()
    request.Message = "Request to " + r.URL.Path
    request.Page = r.URL.Path
    go request.logRequest(logBucket)

})

http.ListenAndServe(":8080",nil)

}
```

Using the log4go package for robust logging

As with most things in Go, where there's something satisfactory and extensible in the core page, it can be taken to the next level by a third party—Go's wonderful logging package is truly brought to life with **log4go**.

Using log4go greatly simplifies the process of file logging, console logging, and logging via TCP/UDP.



For more information on log4go, visit <https://code.google.com/p/log4go/>.

Each instance of a log4go Logger interface can be configured by an XML configuration file and can have filters applied to it to dictate where messaging goes. Let's look at a simple HTTP server to show how we can direct specific logs to location, as shown in the following code:

```
package main

import (
    "code.google.com/p/log4go"
    "net/http"
    "fmt"
    "github.com/gorilla/mux"
)
```

```

var errorLog log4go.Logger
var errorLogWriter log4go.FileLogWriter

var accessLog log4go.Logger
var accessLogWriter *log4go.FileLogWriter

var screenLog log4go.Logger

var networkLog log4go.Logger

```

In the preceding code, we created four distinct log objects — one that writes errors to a logfile, one that writes accesses (page requests) to a separate file, one that sends directly to console (for important notices), and one that passes a log message across the network.

The last two obviously do not need `FileLogWriter`, although it's entirely possible to replicate the network logging using a shared drive if we can mitigate issues with concurrent access, as shown in the following code:

```

func init() {
    fmt.Println("Web Server Starting")
}

func pageHandler(w http.ResponseWriter, r *http.Request) {
    pageFoundMessage := "Page found: " + r.URL.Path
    accessLog.Info(pageFoundMessage)
    networkLog.Info(pageFoundMessage)
    w.Write([]byte("Valid page"))
}

```

Any request to a valid page goes here, sending the message to the `web-access.log` file `accessLog`.

```

func notFound(w http.ResponseWriter, r *http.Request) {
    pageNotFoundMessage := "Page not found / 404: " + r.URL.Path
    errorLog.Info(pageNotFoundMessage)
    w.Write([]byte("Page not found"))
}

```

As with the `accessLog` file, we'll take any 404 / page not found request and route it directly to the `notFound()` method, which saves a fairly generic error message along with the invalid / missing URL requested. Let's look at what we'll do with extremely important errors and messages in the following code:

```

func restricted(w http.ResponseWriter, r *http.Request) {
    message := "Restricted directory access attempt!"
}

```

```
    errorLog.Info(message)
    accessLog.Info(message)
    screenLog.Info(message)
    networkLog.Info(message)
    w.Write([]byte("Restricted!"))
}
```

The `restricted()` function and corresponding `screenLog` represents a message we deem as *critical* and worthy of going to not only the error and the access logs, but also to screen and passed across the wire as a `networkLog` item. In other words, it's a message so important that everybody gets it.

In this case, we're detecting attempts to get at our `.git` folder, which is a fairly common accidental security vulnerability that people have been known to commit in automatic file uploads and updates. Since we have cleartext passwords represented in files and may expose that to the outside world, we'll catch this on request and pass to our critical and noncritical logging mechanisms.

We might also look at this as a more open-ended bad request notifier — one worthy of immediate attention from a network developer. In the following code, we'll start creating a few loggers:

```
func main() {

    screenLog = make(log4go.Logger)
    screenLog.AddFilter("stdout", log4go.DEBUG, log4go.
NewConsoleLogWriter())

    errorLogWriter := log4go.NewFileLogWriter("web-errors.log",
false)
    errorLogWriter.SetFormat("%d %t - %M (%S)")
    errorLogWriter.SetRotate(false)
    errorLogWriter.SetRotateSize(0)
    errorLogWriter.SetRotateLines(0)
    errorLogWriter.SetRotateDaily(true)
```

Since `log4go` opens up a bevy of additional logging options, we can play a bit with how our logs rotate and are formatted without having to draw that out specifically with `Sprintf` or something similar.

The options here are simple and expressive:

- `SetFormat`: This allows us to specify how our individual log lines will look.
- `SetRotate`: This permits automatic rotation based on the size of the file and/or the number of lines in log. The `SetRotateSize()` option sets rotation on bytes in the message and `SetRotateLines()` sets the maximum number of lines. The `SetRotateDaily()` function lets us create new logfiles based on the day regardless of our settings in the previous functions. This is a fairly common logging technique and can generally be burdensome to code by hand.

The output of our logging format ends up looking like the following line of code:

```
04/13/14 10:46 - Page found%(EXTRA string=/valid)
(main.pageHandler:24)
```

The `%S` part is the source, and that gives us the line number and our method trace for the part of our application that invoked the log:

```
errorLog = make(log4go.Logger)
errorLog.AddFilter("file", log4go.DEBUG, errorLogWriter)

networkLog = make(log4go.Logger)
networkLog.AddFilter("network", log4go.DEBUG,
    log4go.NewSocketLogWriter("tcp", "localhost:3000"))
```

Our network log sends JSON-encoded messages via TCP to the address we provide. We'll show a very simple handling server for this in the next section of code that translates the log messages into a centralized logfile:

```
accessLogWriter = log4go.NewFileLogWriter("web-access.log", false)
accessLogWriter.SetFormat("%d %t - %M (%S)")
accessLogWriter.SetRotate(true)
accessLogWriter.SetRotateSize(0)
accessLogWriter.SetRotateLines(500)
accessLogWriter.SetRotateDaily(false)
```

Our `accessLogWriter` is similar to the `errorLogWriter` except that instead of rotating daily, we rotate it every 500 lines. The idea here is that access logs would of course be more frequently touched than an error log—hopefully. The code is as follows:

```
accessLog = make(log4go.Logger)
accessLog.AddFilter("file", log4go.DEBUG, accessLogWriter)

rtr := mux.NewRouter()
rtr.HandleFunc("/valid", pageHandler)
```



```
rtr.HandleFunc("/.git/", restricted)
rtr.NotFoundHandler = http.HandlerFunc(notFound)
```

In the preceding code, we used the Gorilla Mux package for routing. This gives us easier access to the 404 handler, which is less than simplistic to modify in the basic http package built directly into Go. The code is as follows:

```
http.Handle("/", rtr)
http.ListenAndServe(":8080", nil)
}
```

Building the receiving end of a network logging system like this is also incredibly simple in Go, as we're building nothing more than another TCP client that can handle the JSON-encoded messages.

We can do this with a receiving server that looks remarkably similar to our TCP chat server from an earlier chapter. The code is as follows:

```
package main

import (
    "net"
    "fmt"
)

type Connection struct {

}

func (c Connection) Listen(l net.Listener) {
    for {
        conn, _ := l.Accept()
        go c.logListen(conn)
    }
}
```

As with our chat server, we bind our listener to a Connection struct, as shown in the following code:

```
func (c *Connection) logListen(conn net.Conn) {
    for {
        buf := make([]byte, 1024)
        n, _ := conn.Read(buf)
        fmt.Println("Log Message", string(n))
    }
}
```

In the preceding code, we receive log messages delivered via JSON. At this point, we're not unmarshalling the JSON, but we've shown how to do that in an earlier chapter.

Any message sent will be pushed into the buffer—for this reason, it may make sense to expand the buffer's size depending on how detailed the information is.

```
func main() {
    serverClosed := make(chan bool)

    listener, err := net.Listen("tcp", ":3000")
    if err != nil {
        fmt.Println ("Could not start server!",err)
    }

    Conn := Connection{}

    go Conn.Listen(listener)

    <-serverClosed
}
```

You can imagine how network logging can be useful, particularly in server clusters where you might have a selection of, say, web servers and you don't want to reconcile individual logfiles into a single log.

Panicking

With all the discussion of capturing errors and logging them, we should probably consider the `panic()` and `recover()` functionality in Go.

As briefly discussed earlier, `panic()` and `recover()` operate as a more basic, immediate, and explicit error detection methodology than, say, `try/catch/finally` or even Go's built-in error return value convention. As designed, `panic()` unwinds the stack and leads to program exit unless `recover()` is invoked. This means that unless you explicitly recover, your application will end.

So, how is this useful other than for stopping execution? After all, we can catch an error and simply end the application manually through something similar to the following code:

```
package main

import
(
    "fmt"
```

```
    "os"
)

func processNumber(un int) {

    if un < 1 || un > 4 {
        fmt.Println("Now you've done it!")
        os.Exit(1)
    }else {
        fmt.Println("Good, you can read simple instructions.")
    }
}

func main() {
    userNum := 0
    fmt.Println("Enter a number between 1 and 4.")
    _,err := fmt.Scanf("%d",&userNum)
    if err != nil {}

    processNumber(userNum)
}
```

However, while this function does sanity checking and enacts a permanent, irreversible application exit, `panic()` and `recover()` allow us to reflect errors from a specific package and/or method, save those, and then resume gracefully.

This is very useful when we're dealing with methods that are called from other methods that are called from other methods, and so on. The types of deeply embedded or recursive functions that make it hard to discern a specific error are where `panic()` and `recover()` are most advantageous. You can also imagine how well this functionality can play with logging.

Recovering

The `panic()` function on its own is fairly simple, and it really becomes useful when paired with `recover()` and `defer()`.

Take, for example, an application that returns meta information about a file from the command line. The main part of the application will listen for user input, pass this into a function that will open the file, and then pass that file reference to another function that will get the file's details.

Now, we can obviously stack errors as return elements straight through the process, or we can panic along the way, recover back down the steps, and gather our errors at the bottom for logging and/or reporting directly to console.

Avoiding spaghetti code is a welcomed side effect of this approach versus the former one. Think of this in a general sense (this is pseudo code):

```
func getFileDetails(fileName string) error {
    return err
}

func openFile(fileName string) error {
    details,err := getFileDetails(fileName)
    return err
}

func main() {

    file,err := openFile(fileName)

}
```

With a single error, it's entirely manageable to approach our application in this way. However, when each individual function has one or more points of failure, we will require more and more return values and a way of reconciling them all into a single overall error message or messages. Check the following code:

```
package main

import
(
    "os"
    "fmt"
    "strconv"
)

func gatherPanics() {
    if rec := recover(); rec != nil {
        fmt.Println("Critical Error:", rec)
    }
}
```

This is our general recovery function, which is called before every method on which we wish to capture any panic. Let's look at a function to deduce the file's details:

```
func getFileDetails(fileName string) {
    defer gatherPanics()
    finfo, err := os.Stat(fileName)
    if err != nil {
        panic("Cannot access file")
    } else {
        fmt.Println("Size: ", strconv.FormatInt(finfo.Size(), 10))
    }
}

func openFile(fileName string) {
    defer gatherPanics()
    if _, err := os.Stat(fileName); err != nil {
        panic("File does not exist")
    }
}
```

The two functions from the preceding code are merely an attempt to open a file and panic if the file does not exist. The second method, `getFileDetails()`, is called from the `main()` function such that it will always execute, regardless of a blocking error in `openFile()`.

In the real world, we will often develop applications where a nonfatal error stops just a portion of the application from working, but will not cause the application as a whole to break. Check the following code:

```
func main() {
    var fileName string
    fmt.Print("Enter filename>")
    _, err := fmt.Scanf("%s", &fileName)
    if err != nil {}
    fmt.Println("Getting info for", fileName)

    openFile(fileName)
    getFileDetails(fileName)
}
```

If we were to remove the `recover()` code from our `gatherPanics()` method, the application would crash if/when the file didn't exist.

This may seem ideal, but imagine a scenario where a user selects a nonexistent file for a directory that they lack the rights to view. When they solve the first problem, they will be presented with the second instead of seeing all potential issues at one time.

The value of expressive errors can't be overstated from a user experience standpoint. Gathering and presenting expressive errors is made easier through this methodology—even a `try/catch/finally` requires that we (as developers) explicitly do something with the returned error in the catch clause.

Logging our panics

In the preceding code, we can integrate a logging mechanism pretty simply in addition to catching our panics.

One consideration about logging that we haven't discussed is the notion of when to log. As our previous examples illustrate, we can sometimes run into problems that should be logged but may be mitigated by future user action. As such, we can choose to log our errors immediately or save it until the end of execution or a greater function.

The primary benefit of logging immediately is that we're not susceptible to an actual crash preventing our log from being saved. Take the following example:

```
type LogItem struct {
    Message string
    Function string
}
```

```
var Logs []LogItem
```

We've created a log struct and a slice of LogItems using the following code:

```
func SaveLogs() {
    logFile := log4go.NewFileLogWriter("errors.log", false)
    logFile.SetFormat("%d %t - %M (%S)")
    logFile.SetRotate(true)
    logFile.SetRotateSize(0)
    logFile.SetRotateLines(500)
    logFile.SetRotateDaily(false)

    errorLog := make(log4go.Logger)
    errorLog.AddFilter("file", log4go.DEBUG, logFile)
```

```
    for i:= range Logs {
        errorLog.Info(Logs[i].Message + " in " + Logs[i].Function)
    }

}
```

This, ostensibly, is where all of our captured `LogItems` will be turned into a good collection of line items in a logfile. There is a problem, however, as illustrated in the following code:

```
func registerError(block chan bool) {

    Log := LogItem{ Message:"An Error Has Occurred!", Function:
        "registerError()" }
    Logs = append(Logs,Log)
    block <- true
}
```

Executed in a goroutine, this function is non-blocking and allows the main thread's execution to continue. The problem is with the following code that runs after the goroutine, which causes us to log nothing at all:

```
func separateFunction() {
    panic("Application quitting!")
}
```

Whether invoked manually or by the binary itself, the application quitting prematurely precludes our logfiles from being written, as that method is deferred until the end of the `main()` method. The code is as follows:

```
func main() {
    block := make(chan bool)
    defer SaveLogs()
    go func(block chan bool) {

        registerError(block)

    }(block)

    separateFunction()

}
```

The tradeoff here, however, is performance. If we execute a file operation every time we want to log something, we're potentially introducing a bottleneck into our application. In the preceding code, errors are sent via goroutine but written in blocking code—if we introduce the log writing directly into `registerError()`, it can slow down our final application.

As mentioned previously, one opportunity to mitigate these issues and allow the application to still save all of our log entries is to utilize either memory logging or network logging.

Catching stack traces with concurrent code

In earlier Go releases, the ability to properly execute a stack trace from our source was a daunting task, which is emblematic of some of the many complaints and concerns users had early on about general error handling in Go.

While the Go team has remained vigilant about the *right* way to do this (as they have with several other key language features such as a lack of generics), stack traces and stack info have been tweaked a bit as the language has grown.

Using the runtime package for granular stack traces

In an effort to capture stack traces directly, we can glean some helpful pieces of information from the built-in runtime package.

Specifically, Go provides a couple of tools to give us insight into the invocation and/or breakpoints of a goroutine. The following are the functions within the runtime package:

- `runtime.Caller()`: This returns information about the parent function of a goroutine
- `runtime.Stack()`: This allocates a buffer for the amount of data in a stack trace and then fills that with the trace
- `runtime.NumGoroutine()`: This returns the total number of open goroutines

We can utilize all three preceding tools to better describe the inner workings of any given goroutine and related errors.

Using the following code, we'll spawn some random goroutines doing random things and log not only the goroutine's log message, but also the stack trace and the goroutine's caller:

```
package main

import (
    "os"
    "fmt"
    "runtime"
    "strconv"
    "code.google.com/p/log4go"
)

type LogItem struct {
    Message string
}

var LogItems []LogItem

func saveLogs() {
    logFile := log4go.NewFileLogWriter("stack.log", false)
    logFile.SetFormat("%d %t - %M (%S)")
    logFile.SetRotate(false)
    logFile.SetRotateSize(0)
    logFile.SetRotateLines(0)
    logFile.SetRotateDaily(true)

    logStack := make(log4go.Logger)
    logStack.AddFilter("file", log4go.DEBUG, logFile)
    for i := range LogItems {
        fmt.Println(LogItems[i].Message)
        logStack.Info(LogItems[i].Message)
    }
}
```

The `saveLogs()` function merely takes our map of `LogItems` and applies them to file per `log4go`, as we did earlier in the chapter. Next, we'll look at the function that supplies details on our goroutines:

```
func goDetails(done chan bool) {
    i := 0
    for {
        var message string
        stackBuf := make([]byte, 1024)
        stack := runtime.Stack(stackBuf, false)
        stack++
        _, callerFile, callerLine, ok := runtime.Caller(0)
        message = "Goroutine from " + string(callerLine) + " " +
            string(callerFile) + " stack:" + string(stackBuf)
        openGoroutines := runtime.NumGoroutine()

        if (ok == true) {
            message = message + callerFile
        }

        message = message +
            strconv.FormatInt(int64(openGoroutines), 10) + " goroutines
            active"

        li := LogItem{ Message: message }

        LogItems = append(LogItems, li)
        if i == 20 {
            done <- true
            break
        }

        i++
    }
}
```

This is where we gather more details about a goroutine. The `runtime.Caller()` function provides a few returned values: its pointer, the filename of the caller, the line of the caller. The last return value indicates whether the caller could be found.

As mentioned previously, `runtime.NumGoroutine()` gives us the number of extant goroutines that have not yet been closed.

Then, in `runtime.Stack(stackBuf, false)`, we fill our buffer with the stack trace. Note that we're not trimming this byte array to length.

All three are passed into `LogItem.Message` for later use. Let's look at the setup in the `main()` function:

```
func main() {
    done := make(chan bool)

    go goDetails(done)
    for i:= 0; i < 10; i++ {
        go goDetails(done)
    }


    for {
        select {
            case d := <-done:
                if d == true {
                    saveLogs()
                    os.Exit(1)
                }
        }
    }
}
```

Finally, we loop through some goroutines that are doing loops themselves and exit upon completion.

When we examine our logfile, we're given far more verbose details on our goroutines than we have previously, as shown in the following code:

```
04/16/14 23:25 - Goroutine from + /var/log/go/ch9_11_stacktrace.
goch9_11_stacktrace.go stack:goroutine 4 [running]:
main.goDetails(0xc08400b300)
    /var/log/go/ch9_11_stacktrace.goch9_11_stacktrace.go:41 +0x8e
created by main.main
    /var/log/go/ch9_11_stacktrace.goch9_11_stacktrace.go:69 +0x4c

/var/log/go/ch9_11_stacktrace.goch9_11_stacktrace.go14 goroutines
active (main.saveLogs:31)
```

 For more information on the runtime package,
go to <http://golang.org/pkg/runtime/>.

Summary

Debugging, testing, and logging concurrent code can be particularly cumbersome, often when concurrent goroutines fail in a seemingly silent fashion or fail to execute whatsoever.

We looked at various methods of logging, from file to console to memory to network logging, and examined how concurrent application pieces can fit into these various implementations.

By now, you should be comfortable and natural in creating robust and expressive logs that rotate automatically, impose no latency or bottlenecks, and assist in debugging your applications.

You should feel comfortable with the basics of the runtime package. We'll dive into the testing package, controlling goroutines more explicitly, and unit testing as we dig deeper in the next chapter.

In addition to further examining the testing and runtime packages, in our final chapter, we'll also broach the topic of more advanced concurrency topics in Go as well as review some overall best practices as they relate to programming in the Go language.

10

Advanced Concurrency and Best Practices

Once you're comfortable with the basic and intermediate usage of concurrency features in Go, you may find that you're able to handle the majority of your development use cases with bidirectional channels and standard concurrency tools.

In *Chapter 2, Understanding the Concurrency Model*, and *Chapter 3, Developing a Concurrent Strategy*, we looked at the concurrency models, not just of Go but of other languages as well, and compared the way they – and distributed models – can work. In this chapter, we'll touch on those and some higher level concepts with regard to designing and managing your concurrent application.

In particular, we're going to look at central management of goroutines and their associated channels – out of the box you may find goroutines to be a set-it-and-forget-it proposition; however, there are cases where we might want more granular control of a channel's state.

We've also looked quite a bit at testing and benchmarking from a high level, but we'll look at some more detailed and complex methods for testing. We'll also explore a primer on the Google App Engine, which will give us access to some specific testing tools we haven't yet used.

Finally, we'll touch upon some general best practices for Go, which will surely pertain not just to concurrent application design but your future work in general with the language.

Going beyond the basics with channels

We've talked about quite a few different channel implementations—channels of different type (interfaces, functions, structs, and channels)—and touched upon the differences in buffered and unbuffered channels. However, there's still a lot more we can do with the design and flow of our channels and goroutines.

By design, Go wants you to keep things simple. And that's fantastic for 90 percent of what you'll do with Go. But there are other times where you'll need to dig a little deeper for a solution, or when you'll need to save resources by preserving the amount of open goroutine processes, channels, and more.

You may, at some point, want some hands on control of the size and state, and also the control of a running or closed goroutine, so we'll look at doing that.

Just as importantly, designing your goroutines to work in concert with the application design as a whole can be critical to unit testing, which is a topic we'll touch on in this final chapter.

Building workers

Earlier in this book, we talked about concurrency patterns and a bit about workers. We even brought the workers concept into play in the previous chapter, when we were building our logging systems.

Truly speaking, "worker" is a fairly generic and ambiguous concept, not just in Go, but in general programming and development. In some languages, it's an object/instantiated class, and in others it's a concurrent actor. In functional programming languages, worker is a graduated function return passed to another.

If we go back to the preface, we will see that we have literally used the Go gopher as an example of a worker. In short, a worker is something more complex than a single function call or programmatic action that will perform a task one or more times.

So why are we talking about it now? When we build our channels, we are creating a mechanism to do work. When we have a struct or an interface, we're combining methods and values at a single place, and then doing work using that *object* as both a mechanism for the work as well as a place to store information about that work.

This is particularly useful in application design, as we're able to delegate various elements of an application's functionality to distinct and well-defined workers. Consider, for example, a server pinging application that has specific pieces doing specific things in a self-contained, compartmentalized manner.

We'll attempt to check for server availability via the HTTP package, check the status code and errors, and back off if we find problems with any particular server. You can probably see where this is going – this is the most basic approach to load balancing. But an important design consideration is the way in which we manage our channels.

We'll have a master channel, where all important global transactions should be accumulated and evaluated, but each individual server will also have its own channels for handling tasks that are important only to that individual struct.

The design in the following code can be considered as a rudimentary pipeline, which is roughly akin to the producer/consumer model we talked about in the previous chapters:

```
package main

import (
    "fmt"
    "time"
    "net/http"
)

const INIT_DELAY = 3000
const MAX_DELAY = 60000
const MAX_RETRIES = 4
const DELAY_INCREMENT = 5000
```

The preceding code gives the configuration part of the application, setting scope on how frequently to check servers, the maximum amount of time for backing off, and the maximum amount of retries before giving up entirely.

The `DELAY_INCREMENT` value represents how much time we will add to our server checking process each time we discover a problem. Let's take a look at how to create a server in the following section:

```
var Servers []Server

type Server struct {
    Name string
    URI string
    LastChecked time.Time
    Status bool
    StatusCode int
    Delay int
    Retries int
    Channel chan bool
}
```


Now, we design the basic server (using the following code), which contains its current status, the last time it was checked, the present delay between checks, its own channel for evaluating statuses and establishing the new status, and updated retry delay:

```
func (s *Server) checkServerStatus(sc chan *Server) {
    var previousStatus string

    if s.Status == true {
        previousStatus = "OK"
    } else {
        previousStatus = "down"
    }

    fmt.Println("Checking Server", s.Name)
    fmt.Println("\tServer was", previousStatus, "on last check
        at", s.LastChecked)

    response, err := http.Get(s.URI)
    if err != nil {
        fmt.Println("\tError: ", err)
        s.Status = false
        s.StatusCode = 0
    } else {
        fmt.Println(response.Status)
        s.StatusCode = response.StatusCode
        s.Status = true
    }

    s.LastChecked = time.Now()
    sc <- s
}
```

The `checkServerStatus()` method is the meat and potatoes of our application here. We pass all of our servers through this method in the `main()` function to our `cycleServers()` loop, after which it becomes self-fulfilling.

If our `Status` is set to `true`, we send the state to the console as `OK` (otherwise `down`) and set our `Server` status code with `s.StatusCode` as either the HTTP code or 0 if there was a network or other error.

Finally, set the last-checked time of `Server` to `Now()` and pass `Server` through the `serverChan` channel. In the following code, we'll demonstrate how we'll rotate through our available servers:

```
func cycleServers(sc chan *Server) {

    for i := 0; i < len(Servers); i++ {
        Servers[i].Channel = make(chan bool)
        go Servers[i].updateDelay(sc)
        go Servers[i].checkServerStatus(sc)
    }

}
```

This is our initial loop, called from `main`. It simply loops through our available servers and initializes its listening goroutine as well as sending the first `checkServerStatus` request.

It's worth noting two things here: first, the channel invoked by `Server` will never actually die, but instead the application will stop checking the server. That's fine for all practical purposes here, but if we have thousands and thousands of servers to check, we're wasting resources on what essentially amounts to an unclosed channel and a map element that has not been removed. Later, we'll broach the concept of manually killing goroutines, something we've only been able to do through abstraction by stopping the communication channel. Let's now take a look at the following code that controls a server's status and its next steps:

```
func (s *Server) updateDelay(sc chan *Server) {
    for {
        select {
            case msg := <- s.Channel:

                if msg == false {
                    s.Delay = s.Delay + DELAY_INCREMENT
                    s.Retries++
                    if s.Delay > MAX_DELAY {
                        s.Delay = MAX_DELAY
                    }
                }
            }
        }
    }
}
```

```
    }else {
        s.Delay = INIT_DELAY
    }
    newDuration := time.Duration(s.Delay)

    if s.Retries <= MAX_RETRIES {
        fmt.Println("\tWill check server again")
        time.Sleep(newDuration * time.Millisecond)
        s.checkServerStatus(sc)
    }else {
        fmt.Println("\tServer not reachable
        after",MAX_RETRIES,"retries")
    }

    default:
}
}
```

This is where each `Server` will listen for changes in its status, as reported by `checkServerStatus()`. When any given `Server` struct receives a message that a change in status has been reported via our initial loop, it will evaluate that message and act accordingly.

If the `Status` is set to `false`, we know that the server was inaccessible for some reason. The `Server` reference itself will then add a delay to the next time it's checked. If it's set to `true`, the server was accessible and the delay will either be set or reset to the default retry value of `INIT_DELAY`.

It finally sets a sleep mode on that goroutine before reinitializing the `checkServerStatus()` method on itself, passing the `serverChan` reference along in the initial goroutine loop in the `main()` function:

```
func main() {

    endChan := make(chan bool)
    serverChan := make(chan *Server)

    Servers = []Server{ {Name: "Google", URI: "http://www.google.com",
    Status: true, Delay: INIT_DELAY}, {Name: "Yahoo", URI: "http://www.
    yahoo.com", Status: true, Delay: INIT_DELAY}, {Name: "Bad Amazon",
    URI: "http://amazon.zom", Status: true, Delay: INIT_DELAY} }
```

One quick note here—in our slice of `Servers`, we intentionally introduced a typo in the last element. You'll notice `amazon.zom`, which will provoke an HTTP error in the `checkServerStatus()` method. The following is the function to cycle through servers to find an appropriate match:

```

    go cycleServers(serverChan)

    for {
        select {
            case currentServer := <- serverChan:
                currentServer.Channel <- false
            default:
        }
    }

    <- endChan
}

```

The following is an example of the output with the typo included:

Checking Server Google

```

    Server was OK on last check at 0001-01-01 00:00:00 +0000 UTC
    200 OK
    Will check server again

```

Checking Server Yahoo

```

    Server was OK on last check at 0001-01-01 00:00:00 +0000 UTC
    200 OK
    Will check server again

```

Checking Server Amazon

```

    Server was OK on last check at 0001-01-01 00:00:00 +0000 UTC
    Error:  Get http://amazon.zom: dial tcp: GetAddrInfoW: No such
host is known.
    Will check server again

```

Checking Server Google

```

    Server was OK on last check at 2014-04-23 12:49:45.6575639 -0400
EDT

```

We'll be taking the preceding code for one last spin through some concurrency patterns later in this chapter, turning it into something a bit more practical.

Implementing nil channel blocks

One of the bigger problems in designing something like a pipeline or producer/consumer model is there's somewhat of a black hole when it comes to the state of any given goroutine at any given time.

Consider the following loop, wherein a producer channel creates an arbitrary set of consumer channels and expects each to do one and only one thing:

```
package main

import (
    "fmt"
    "time"
)

const CONSUMERS = 5

func main() {

    Producer := make(chan (chan int))

    for i := 0; i < CONSUMERS; i++ {
        go func() {
            time.Sleep(1000 * time.Microsecond)
            conChan := make(chan int)

            go func() {
                for {
                    select {
                        case _, ok := <-conChan:
                            if ok {
                                Producer <- conChan
                            } else {
                                return
                            }
                        default:
                    }
                }
            }()

            conChan <- 1
            close(conChan)
        }()
    }
}
```

Given a random amount of consumers to produce, we attach a channel to each and pass a message upstream to the `Producer` via that consumer's channel. We send just a single message (which we could handle with a buffered channel), but we simply close the channel after.

Whether in a multithreaded application, a distributed application, or a highly concurrent application, an essential attribute of a producer-consumer model is the ability for data to move across a queue/channel in a steady, reliable fashion. This requires some modicum of mutual knowledge to be shared between both the producer and consumers.

Unlike environments that are distributed (or multicore), we do possess some inherent awareness of the status on both ends of that arrangement. We'll next look at a listening loop for producer messages:

```
for {
    select {
    case consumer, ok := <-Producer:
        if ok == false {
            fmt.Println("Goroutine closed?")
            close(Producer)
        } else {
            log.Println(consumer)
            // consumer <- 1
        }
        fmt.Println("Got message from secondary channel")
    default:
    }
}
```

The primary issue is that one of the `Producer` channel doesn't know much about any given `Consumer`, including when it's actively running. If we uncommented the `// consumer <- 1` line, we'll get a panic, because we're attempting to send a message on a closed channel.

As a message is passed across a secondary goroutine's channel, upstream to the channel of the `Producer`, we get an appropriate reception, but cannot detect when the downstream goroutine is closed.

Knowing when a goroutine has terminated is in many cases inconsequential, but consider an application that spawns new goroutines when a certain number of tasks are complete, effectively breaking a task into mini tasks. Perhaps each chunk is dependent on the total completion of the last chunk, and a broadcaster must know the status of the current goroutines before moving on.

Using nil channels

In the earlier versions of Go, you could communicate across uninitialized, thus nil or 0-value channels without a panic (although your results would be unpredictable). Starting from Go Version 1, communication across nil channels produced a consistent but sometimes confusing effect.

It's vital to note that within a select switch, transmission on a nil channel on its own will still cause a deadlock and panic. This is something that will most often creep up when utilizing global channels and not ever properly initializing them. The following is an example of such transmission on a nil channel:

```
func main() {  
  
    var channel chan int  
  
    channel <- 1  
  
    for {  
        select {  
            case <- channel:  
  
            default:  
        }  
    }  
}
```

As the channel is set to its 0 value (nil, in this case), it blocks perpetually and the Go compiler will detect this, at least in more recent versions. You can also duplicate this outside of a select statement, as shown in the following code:

```
var done chan int  
defer close(done)  
defer log.Println("End of script")  
go func() {  
    time.Sleep(time.Second * 5)  
    done <- 1  
}()  
  
for {  
    select {  
        case <- done:  
            log.Println("Got transmission")  
            return  
        default:  
    }  
}
```

The preceding code will block forever without the panic, due to the default in the `select` statement keeping the main loop active while waiting for communication on the channel. If we initialize the channel, however, the application runs as expected.

With these two fringe cases—closed channels and nil channels—we need a way for a master channel to understand the state of a goroutine.

Implementing more granular control over goroutines with tomb

As with many such problems—both niche and common—there exists a third-party utility for grabbing your goroutines by the horns.

Tomb is a library that provides diagnostics to go along with any goroutine and channel—it can tell a master channel if another goroutine is dead or dying.

In addition, it allows you to explicitly kill a goroutine, which is a bit more nuanced than simply closing the channel it is attached to. As previously mentioned, closing the channel is effectively neutering a goroutine, although it could ultimately still be active.

You are about to find a simple fetch-and-grab body script that takes a slice of URL structs (with status and URI) and attempts to grab the HTTP response for each and apply it to the struct. But instead of just reporting information from the goroutines, we'll have the ability to send "kill messages" to each of a "master" struct's child goroutines.

In this example, we'll run the script for 10 seconds, and if any of the goroutines fail to do their job in that allotted time, it will respond that it was unable to get the URL's body due to a kill send from the master struct that invoked it:

```
package main

import (
    "fmt"
    "io/ioutil"
    "launchpad.net/tomb"
    "net/http"
    "strconv"
    "sync"
    "time"
)

var URLs []URL

type GoTomb struct {
    tomb tomb.Tomb
}
```


This is the minimum necessary structure required to create a parent or a master struct for all of your spawned goroutines. The `tomb.Tomb` struct is simply a mutex, two channels (one for dead and dying), and a reason error struct. The structure of the `URL` struct looks like the following code:

```
type URL struct {
    Status bool
    URI    string
    Body   string
}
```

Our `URL` struct is fairly basic—`Status`, set to `false` by default and `true` when the body has been retrieved. It consists of the `URI` variable—which is the reference to the `URL`—and the `Body` variable for storing the retrieved data. The following function allows us to execute a "kill" on a `GoTomb` struct:

```
func (gt GoTomb) Kill() {

    gt.tomb.Kill(nil)

}
```

The preceding method invokes `tomb.Kill` on our `GoTomb` struct. Here, we have set the sole parameter to `nil`, but this can easily be changed to a more descriptive error, such as `errors.New("Time to die, goroutine")`. Here, we'll show the listener for the `GoTomb` struct:

```
func (gt *GoTomb) TombListen(i int) {

    for {
        select {
        case <-gt.tomb.Dying():
            fmt.Println("Got kill command from tomb!")
            if URLS[i].Status == false {
                fmt.Println("Never got data for", URLS[i].URI)
            }
            return
        }
    }
}
```

We invoke `TombListen` attached to our `GoTomb`, which sets a select that listens for the `Dying()` channel, as shown in the following code:

```
func (gt *GoTomb) Fetch() {
    for i := range URLs {
        go gt.TombListen(i)

        go func(ii int) {

            timeDelay := 5 * ii
            fmt.Println("Waiting ", strconv.FormatInt(int64(timeDelay),
                10), " seconds to get", URLs[ii].URI)
            time.Sleep(time.Duration(timeDelay) * time.Second)
            response, _ := http.Get(URLs[ii].URI)
            URLs[ii].Status = true
            fmt.Println("Got body for ", URLs[ii].URI)
            responseBody, _ := ioutil.ReadAll(response.Body)
            URLs[ii].Body = string(responseBody)
        }(i)
    }
}
```

When we invoke `Fetch()`, we also set the tomb to `TombListen()`, which receives those "master" messages across all spawned goroutines. We impose an intentionally long wait to ensure that our last few attempts to `Fetch()` will come after the `Kill()` command. Finally, our `main()` function, which handles the overall setup:

```
func main() {

    done := make(chan int)

    URLs = []URL{{Status: false, URI: "http://www.google.com", Body:
        ""}, {Status: false, URI: "http://www.amazon.com", Body: ""}, {Status:
        false, URI: "http://www.ubuntu.com", Body: ""}}

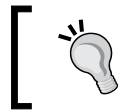
    var MasterChannel GoTomb
    MasterChannel.Fetch()

    go func() {

        time.Sleep(10 * time.Second)
        MasterChannel.Kill()
        done <- 1
    }()
}
```

```
for {
    select {
        case <-done:
            fmt.Println("")
            return
        default:
    }
}
```

By setting `time.Sleep` to 10 seconds and then killing our goroutines, we guarantee that the 5 second delays between `Fetch()` prevent the last of our goroutines from successfully finishing before being killed.



For the `tomb` package, go to <http://godoc.org/launchpad.net/tomb> and install it using the `go get launchpad.net/tomb` command.

Timing out with channels

One somewhat critical point with channels and `select` loops that we haven't examined particularly closely is the ability – and often necessity – to kill a `select` loop after a certain timeout.

Many of the applications we've written so far are long-running or perpetually-running, but there are times when we'll want to put a finite time limit on how long goroutines can operate.

The `for { select { } }` switch we've used so far will either live perpetually (with a default case) or wait to be broken from one or more of the cases.

There are two ways to manage interval-based tasks – both as part of the `time` package, unsurprisingly.

The `time.Ticker` struct allows for any given operation after the specified period of time. It provides `C`, a blocking channel that can be used to detect activity sent after that period of time; refer to the following code:

```
package main

import (
    "log"
    "time"
)
```

```
func main() {  
  
    timeout := time.NewTimer(5 * time.Second)  
    defer log.Println("Timed out!")  
  
    for {  
        select {  
            case <-timeout.C:  
                return  
            default:  
        }  
    }  
}
```

We can extend this to end channels and concurrent execution after a certain amount of time. Take a look at the following modifications:

```
package main  
  
import (  
    "fmt"  
    "time"  
)  
  
func main() {  
  
    myChan := make(chan int)  
  
    go func() {  
        time.Sleep(6 * time.Second)  
        myChan <- 1  
    }()  
  
    for {  
        select {  
            case <-time.After(5 * time.Second):  
                fmt.Println("This took too long!")  
                return  
            case <-myChan:  
                fmt.Println("Too little, too late")  
        }  
    }  
}
```

Building a load balancer with concurrent patterns

When we built our server pinging application earlier in this chapter, it was probably pretty easy to imagine taking this to a more usable and valuable space.

Pinging a server is often the first step in a health check for a load balancer. Just as Go provides a usable out-of-the-box web server solution, it also presents a very clean `Proxy` and `ReverseProxy` struct and methods, which makes creating a load balancer rather simple.

Of course, a round-robin load balancer will need a lot of background work, specifically on checking and rechecking as it changes the `ReverseProxy` location between requests. We'll handle these with the goroutines triggered with each request.

Finally, note that we have some dummy URLs at the bottom in the configuration—changing those to production URLs should immediately turn the server that runs this into a working load balancer. Let's look at the main setup for the application:

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "net/http/httputil"
    "net/url"
    "strconv"
    "time"
)

const MAX_SERVER_FAILURES = 10
const DEFAULT_TIMEOUT_SECONDS = 5
const MAX_TIMEOUT_SECONDS = 60
const TIMEOUT_INCREMENT = 5
const MAX_RETRIES = 5
```

In the previous code, we defined our constants, much like we did previously. We have a `MAX_RETRIES`, which limits how many failures we can have, `MAX_TIMEOUT_SECONDS`, which defines the longest amount of time we'll wait before trying again, and our `TIMEOUT_INCREMENT` for changing that value between failures. Next, let's look at the basic construction of our `Server` struct:

```
type Server struct {
    Name          string
    Failures      int
    InService     bool
    Status        bool
    StatusCode    int
    Addr          string
    Timeout       int
    LastChecked   time.Time
    Recheck       chan bool
}
```

As we can see in the previous code, we have a generic `Server` struct that maintains the present state, the last status code, and information on the last time the server was checked.

Note that we also have a `Recheck` channel that triggers the delayed attempt to check the `Server` again for availability. Each Boolean passed across this channel will either remove the server from the available pool or reannounce that it is still in service:

```
func (s *Server) serverListen(serverChan chan bool) {
    for {
        select {
        case msg := <-s.Recheck:
            var statusText string
            if msg == false {
                statusText = "NOT in service"
                s.Failures++
                s.Timeout = s.Timeout + TIMEOUT_INCREMENT
                if s.Timeout > MAX_TIMEOUT_SECONDS {
                    s.Timeout = MAX_TIMEOUT_SECONDS
                }
            } else {
```

```
        if ServersAvailable == false {
            ServersAvailable = true
            serverChan <- true
        }
        statusText = "in service"
        s.Timeout = DEFAULT_TIMEOUT_SECONDS
    }

    if s.Failures >= MAX_SERVER_FAILURES {
        s.InService = false
        fmt.Println("\tServer", s.Name, "failed too many times.")
    } else {
        timeString := strconv.FormatInt(int64(s.Timeout), 10)
        fmt.Println("\tServer", s.Name, statusText, "will check
            again in", timeString, "seconds")
        s.InService = true
        time.Sleep(time.Second * time.Duration(s.Timeout))
        go s.checkStatus()
    }
}
}
```

This is the instantiated method that listens on each server for messages delivered on the availability of a server at any given time. While running a goroutine, we keep a perpetually listening channel open to listen to Boolean responses from `checkStatus()`. If the server is available, the next delay is set to default; otherwise, `TIMEOUT_INCREMENT` is added to the delay. If the server has failed too many times, it's taken out of rotation by setting its `InService` property to false and no longer invoking the `checkStatus()` method. Let's next look at the method for checking the present status of Server:

```
func (s *Server) checkStatus() {
    previousStatus := "Unknown"
    if s.Status == true {
        previousStatus = "OK"
    } else {
        previousStatus = "down"
    }
    fmt.Println("Checking Server", s.Name)
    fmt.Println("\tServer was", previousStatus, "on last check at",
        s.LastChecked)
```

```

    response, err := http.Get(s.Addr)
    if err != nil {
        fmt.Println("\tError: ", err)
        s.Status = false
        s.StatusCode = 0
    } else {
        s.StatusCode = response.StatusCode
        s.Status = true
    }

    s.LastChecked = time.Now()
    s.Recheck <- s.Status
}

```

Our `checkStatus()` method should look pretty familiar based on the server ping example. We look for the server; if it is available, we pass `true` to our `Recheck` channel; otherwise `false`, as shown in the following code:

```

func healthCheck(sc chan bool) {
    fmt.Println("Running initial health check")
    for i := range Servers {
        Servers[i].Recheck = make(chan bool)
        go Servers[i].serverListen(sc)
        go Servers[i].checkStatus()
    }
}

```

Our `healthCheck` function simply kicks off the loop of each server checking (and re-checking) its status. It's run only one time, and initializes the `Recheck` channel via the `make` statement:

```

func roundRobin() Server {
    var AvailableServer Server

    if nextServerIndex > (len(Servers) - 1) {
        nextServerIndex = 0
    }

    if Servers[nextServerIndex].InService == true {
        AvailableServer = Servers[nextServerIndex]
    } else {
        serverReady := false
        for serverReady == false {

```



```
        for i := range Servers {
            if Servers[i].InService == true {
                AvailableServer = Servers[i]
                serverReady = true
            }
        }

    }
}
nextServerIndex++
return AvailableServer
}
```

The `roundRobin` function first checks the next available `Server` in the queue – if that server happens to be down, it loops through the remaining to find the first available `Server`. If it loops through all, it will reset to 0. Let's look at the global configuration variables:

```
var Servers []Server
var nextServerIndex int
var ServersAvailable bool
var ServerChan chan bool
var Proxy *httputil.ReverseProxy
var ResetProxy chan bool
```

These are our global variables – our `Servers` slice of `Server` structs, the `nextServerIndex` variable, which serves to increment the next `Server` to be returned, `ServersAvailable` and `ServerChan`, which start the load balancer only after a viable server is available, and then our `Proxy` variables, which tell our `http` handler where to go. This requires a `ReverseProxy` method, which we'll look at now in the following code:

```
func handler(p *httputil.ReverseProxy) func(http.ResponseWriter,
*http.Request) {
    Proxy = setProxy()
    return func(w http.ResponseWriter, r *http.Request) {

        r.URL.Path = "/"

        p.ServeHTTP(w, r)

    }
}
```

Note that we're operating on a `ReverseProxy` struct here, which is different from our previous forays into serving webpages. Our next function executes the round robin and gets our next available server:

```
func setProxy() *httputil.ReverseProxy {

    nextServer := roundRobin()
    nextURL, _ := url.Parse(nextServer.Addr)
    log.Println("Next proxy source:", nextServer.Addr)
    prox := httputil.NewSingleHostReverseProxy(nextURL)

    return prox
}
```

The `setProxy` function is called after every request, and you can see it as the first line in our handler. Next we have the general listening function that looks out for requests we'll be reverse proxying:

```
func startListening() {
    http.HandleFunc("/index.html", handler(proxy))
    _ = http.ListenAndServe(":8080", nil)
}

func main() {
    nextServerIndex = 0
    ServersAvailable = false
    ServerChan := make(chan bool)
    done := make(chan bool)

    fmt.Println("Starting load balancer")
    Servers = []Server{{Name: "Web Server 01", Addr: "http://www.google.com", Status: false, InService: false}, {Name: "Web Server 02", Addr: "http://www.amazon.com", Status: false, InService: false}, {Name: "Web Server 03", Addr: "http://www.apple.zom", Status: false, InService: false}}

    go healthCheck(ServerChan)

    for {
        select {
        case <-ServerChan:
            Proxy = setProxy()
        }
```

```
        startListening()
        return

    }
}

<-done
}
```

With this application, we have a simple but extensible load balancer that works with the common, core components in Go. Its concurrency features keep it lean and fast, and we wrote it in a very small amount of code using exclusively standard Go.

Choosing unidirectional and bidirectional channels

For the purpose of simplicity, we've designed most of our applications and sample code with bidirectional channels, but of course any channel can be set unidirectionally. This essentially turns a channel into a "read-only" or "write-only" channel.

If you're wondering why you should bother limiting the direction of a channel when it doesn't save any resources or guarantee an issue, the reason boils down to simplicity of code and limiting the potential for panics.

By now we know that sending data on a closed channel results in a panic, so if we have a write-only channel, we'll never accidentally run into that problem in the wild. Much of this can also be mitigated with `WaitGroups`, but in this case that's a sledgehammer being used on a nail. Consider the following loop:

```
const TOTAL_RANDOMS = 100

func concurrentNumbers(ch chan int) {
    for i := 0; i < TOTAL_RANDOMS; i++ {
        ch <- i
    }
}

func main() {

    ch := make(chan int)
```

```
go concurrentNumbers(ch)

for {
    select {
        case num := <- ch:
            fmt.Println(num)
            if num == 98 {
                close(ch)
            }
        default:
    }
}
```

Since we're abruptly closing our `ch` channel one digit before the goroutine can finish, any writes to it cause a runtime error.

In this case, we are invoking a read-only command, but it's in the `select` loop. We can safeguard this a bit more by allowing only specific actions to be sent on unidirectional channels. This application will always work up to the point where in the channel is closed prematurely, one shy of the `TOTAL_RANDOMS` constant.

Using receive-only or send-only channels

When we limit the direction or the read/write capability of our channels, we also reduce the potential for closed channel deadlocks if one or more of our processes inadvertently sends on such a channel.

So the short answer to the question "When is it appropriate to use a unidirectional channel?" is "Whenever you can."

Don't force the issue, but if you can set a channel to read/write only, it may preempt issues down the road.

Using an indeterminate channel type

One trick that can often come in handy, and we haven't yet addressed, is the ability to have what is effectively a typeless channel.

If you're wondering why that might be useful, the short answer is concise code and application design thrift. Often this is a discouraged tactic, but you may find it useful from time to time, especially when you need to communicate one or more disparate concepts across a single channel. The following is an example of an indeterminate channel type:

```
package main

import (

    "fmt"
    "time"
)

func main() {

    acceptingChannel := make(chan interface{})

    go func() {

        acceptingChannel <- "A text message"
        time.Sleep(3 * time.Second)
        acceptingChannel <- false
    }()

    for {
        select {
            case msg := <- acceptingChannel:
                switch typ := msg.(type) {
                    case string:
                        fmt.Println("Got text message",typ)
                    case bool:
                        fmt.Println("Got boolean message",typ)
                        if typ == false {
                            return
                        }
                    default:
                        fmt.Println("Some other type of message")
                }

            default:

        }

    }

    <- acceptingChannel
}
```

Using Go with unit testing

As with many of the basic and intermediate development and deployment requirements you may have, Go comes with a built-in application for handling unit tests.

The basic premise behind testing is that you create your package and then create a testing package to run against the initial application. The following is a very basic example:

```
mathematics.go
package mathematics

func Square(x int) int {

    return x * 3
}
mathematics_test.go
package mathematics

import
(
    "testing"
)

func Test_Square_1(t *testing.T) {
    if Square(2) != 4 {
        t.Error("Square function failed one test")
    }
}
```

A simple Go test in that subdirectory will give you the response you're looking for. While this was admittedly simple – and purposefully flawed – you can probably see how easy it is to break apart your code and test it incrementally. This is enough to do very basic unit tests out of the box.

Correcting this would then be fairly simple – the same test would pass on the following code:

```
func Square(x int) int {

    return x * x
}
```

The testing package is somewhat limited; however, as it provides basic pass/fails without the ability to do assertions. There are two third-party packages that can step in and help in this regard, and we'll explore them in the following sections.

GoCheck

GoCheck extends the basic testing package primarily by augmenting it with assertions and verifications. You'll also get some basic benchmarking utility out of it that works a little more fundamentally than anything you'd need to engineer using Go.



For more details on GoCheck visit <http://labix.org/gocheck> and install it using `go get gopkg.in/check.v1`.

Ginkgo and Gomega

Unlike GoCheck, Ginkgo (and its dependency Gomega) takes a different approach to testing, utilizing the **behavior-driven development (BDD)** model. Behavior-driven development is a general model for making sure your application does what it should at every step, and Ginkgo formalizes that into some easily parseable properties.

BDD tends to complement test-driven development (for example, unit testing) rather than replacement. It seeks to answer a few critical questions about the way people (or other systems) will interact with your application. In that sense, we'll generally describe a process and what we expect from that process in fairly human-friendly terms. The following is a short snippet of such an example:

```
Describe("receive new remote TCP connection", func() {
    Context("user enters a number", func() {
        It("should be an integer", func() {
        })
    })
})
```

This allows testing to be as granular as unit testing, but also expands the way we handle application usage in verbose and explicit behaviors.

If BDD is something you or your organization is interested in, this is a fantastic, mature package for implementing deeper unit testing.



For more information on Ginkgo go to <https://github.com/onsi/ginkgo> and install it using `go get github.com/onsi/ginkgo/ginkgo`.

For more information on dependency, refer to `go get github.com/onsi/gomega`.

Using Google App Engine

If you're unfamiliar with Google App Engine, the short version is it's a cloud environment that allows for simple building and deployment of **Platform-As-A-Service (paas)** solutions.

Compared to a lot of similar solutions, Google App Engine allows you to build and test your applications in a very simple and straightforward way. Google App Engine allows you to write and deploy in Python, Java, PHP, and of course, Go.

For the most part, Google App Engine provides a standard Go installation that makes it easy to dovetail off of the `http` package. But it also gives you a few noteworthy additional packages that are unique to Google App Engine itself:

Package	Description
<code>appengine/memcache</code>	This provides a distributed memcache installation unique to Google App Engine
<code>appengine/mail</code>	This allows you to send e-mails through an SMTP-esque platform
<code>appengine/log</code>	Given your storage may be more ephemeral here, it formalizes a cloud version of the log
<code>appengine/user</code>	This opens both identity and OAuth capabilities
<code>appengine/search</code>	This gives your application the power of Google search on your own data via datastore
<code>appengine/xmpp</code>	This provides Google Chat-like capabilities
<code>appengine/urlfetch</code>	This is a crawler functionality
<code>appengine/aetest</code>	This extends unit testing for Google App Engine

While Go is still considered beta for Google App Engine, you can expect that if anyone was able to competently deploy it in a cloud environment, it would be Google.

Utilizing best practices

The wonderful thing with Go when it comes to best practices is that even if you don't necessarily do everything right, either Go will yell at you or provide you with the tools necessary to fix it.

If you attempt to include code and not use it, or if you attempt to initialize a variable and not use it, Go will stop you. If you want to clean up your code's formatting, Go enables it with `go fmt`.

Structuring your code

One of the easiest things you can do when building a package from scratch is to structure your code directories in an idiomatic way. The standard for a new package would look something like the following code:

```
/projects/  
  thisproject/  
    bin/  
    pkg/  
    src/  
      package/  
        mypackage.go
```

Setting up your Go code like this is not just helpful for your own organization, but allows you to distribute your package more easily.

Documenting your code

For anyone who has worked in a corporate or collaborative coding environment, documentation is sacrosanct. As you may recall earlier, using the `godoc` command allows you to quickly get information about a package at the command line or via an ad hoc localhost server. The following are the two basic ways you may use `godoc`:

Using <code>godoc</code>	Description
<code>godoc fmt</code>	This brings <code>fmt</code> documentation to the screen
<code>godoc -http=:3000</code>	This hosts the documentation on port <code>:3030</code>

Go makes it super easy to document your code, and you absolutely should. By simply adding single-line comments above each identifier (package, type, or function), you'll append that to the contextual documentation, as shown in the following code:

```
// A demo documentation package  
package documentation  
  
// The documentation struct object  
// Chapter int represents a document's chapter  
// Content represents the text of the documentation  
type Documentation struct {  
    Chapter int  
    Content string  
}
```

```
// Display() outputs the content of any given Document by chapter
func (d Documentation) Display() {

}
```

When installed, this will allow anyone to run the `godoc` documentation on your package and get as much detailed information as you're willing to supply.

You'll often see more robust examples of this in the Go core code itself, and it's worth reviewing that to compare your style of documentation to Google's and the Go community's.

Making your code available via go get

Assuming you've kept your code in a manner consistent with the organizational techniques as listed previously, making your code available via code repositories and hosts should be a cinch.

Using GitHub as the standard, here's how we might design our third-party application:

1. Make sure you stick to the previous structural format.
2. Keep your source files under the directory structures they'll live in remotely. In other words, expect that your local structure will reflect the remote structure.
3. Perhaps obviously, commit only the files you wish to share in the remote repository.

Assuming your repository is public, anyone should be able to get (`go get`) and then install (`go install`) your package.

Keeping concurrency out of your packages

One last point that might seem somewhat out of place given the context of the book—if you're building separate packages that will be imported, avoid including concurrent code whenever possible.

This is not a hard-and-fast rule, but when you consider potential usage, it makes sense—let the main application handle the concurrency unless your package absolutely needs it. Doing so will prevent a lot of hidden and difficult-to-debug behavior that may make your library less appealing.

Summary

It is my sincere hope that you've been able to explore, understand, and utilize the depths of Go's powerful abilities with concurrency through this book.

We've gone over a lot, from the most basic, channel-free concurrent goroutines to complex channel types, parallelism, and distributed computing, and we've brought some example code along at every step.

By now, you should be fully equipped to build anything your heart desires in code, in a manner that is highly concurrent, fast, and error-free. Beyond that, you should be able to produce well-formed, properly-structured, and documented code that can be used by you, your organization, or others to implement concurrency where it is best utilized.

Concurrency itself is a vague concept; it's one that means slightly different things to different people (and across multiple languages), but the core goal is always fast, efficient, and reliable code that can provide performance boosts to any application.

Armed with a full understanding of both the implementation of concurrency in Go as well as its inner workings, I hope you continue your Go journey as the language evolves and grows, and similarly implore you to consider contributing to the Go project itself as it develops.

Index

Symbols

`/dynamic/[request][number]` 159
`--gif` flag 146
`--gv` flag 146
`--pdf` flag 146
`-race` flag 71, 76, 143
`/static/[request]` 159
`--SVG` flag 146
`/template/[request]` 159
`--text` flag 146
`--web` flag 146

A

AB, Apache
URL 161
actor model
and CSP, difference between 60
Add() function 246
Add() method 13
addToScrapedText function 37, 39
Apache server
disadvantages 156
Apache Zookeeper
URL 121
App Engine
benefits 191
URL 191
using 191
appengine/aetest 297
appengine/log 297
appengine/mail 297
appengine/memcache 297
appengine/search 297
appengine/urifetch 297

appengine/user 297
appengine/xmpp 297
applicationStatus variable 36
assembly
used, in Go 109-111
asynchronous goroutine
versus synchronous goroutine 42
atomic consistency / mutual exclusion 117

B

backup function 231
backup process, concurrent
application 206, 207
behavior-driven development (BDD) 296
best practices, Go
code availability, via go get 299
code, documenting 298, 299
code, structuring 298
utilizing 297
bidirectional channel
selecting 292, 293
Blocking method 1
serial channel, listening 124
Blocking method 2
select statement, using 140
Blocking method 3
network connections 141
reads 141
blocking methods
Blocking method 1 124
Blocking method 2 139
in Go 124
blocking web server
benchmarking against 160-162
Body variable 282

broadcast function 197

buffered channel 29

bus topology

about 199

advantage 199

disadvantages 199

C

C

implementing 103, 104

C10K

attacking, concurrent connections used 157

C10K problem

another approach 158

concurrency used, to attack C10K 156, 157

handling 154

servers, failing at concurrent

connections 155, 156

C10K web server

blocking web server, benchmarking

against 160-162

building 159

requests, handling 162-165

requests, routing 165, 166

call graph

 182

Cassandra

about 215, 216

features 215

CDNs (Content Delivery Networks) 159

cgo

memory space 104

structure 104, 105

channels

best practices 272

buffered channel 29

complex channels, using 128, 129

creating, of channels 141-143

data types, sending via 125

function channel, creating 126

goroutines, cleaning 29

implementing 22-25

interface channel, using 127

interfaces, using 128, 129

letter capitalization 25-28

select statement, using 29-33

structs, using 128, 129

timing out with 284, 285

unbuffered channel 29

used, for building web spider 35-39

checkServerStatus() method 274, 276

client

about 130, 131

examining 137

Client struct 225

close() method 140

closure

working 34, 35

code

documenting 298

structuring 298

code availability

via go get 299

command-line file revision process,

concurrent application

about 238-240

Go, using in daemons 240

Communicating Sequential Processes

(CSP) 29

complete graph. *See* mesh topology

complex channels

using 128, 129

complex concurrency

efficiency, applying in 67, 68

concurrency

impacting, on I/O pprof 188-190

using 62

visualizing 44-49

concurrent application

backup process 206

command-line file revision process 238-240

configuration files, handling 223, 224

designing 206

file changes, detecting 224-226

file listener 206

files, backing up 231, 232

filesystem changes, monitoring 221

logfiles, managing 222

NoSQL, using 208

requisites, identifying 207, 208

server health, checking 241

web-CLI interface 206

web interface, designing 233-238

- concurrent code**
 - stack traces, catching with 265
- concurrent connections**
 - servers, failing at 155, 156
 - used, to attack C10K 156, 157
- concurrent operations**
 - synchronizing 78
- concurrent pattern**
 - used, for building load balancer 286-292
 - visualizing 81
- configuration files, concurrent application**
 - handling 223, 224
- connection**
 - mitigating 178, 179
- conn.Read() function 131**
- consistency**
 - need for 78
- consistency models**
 - about 113
 - atomic consistency / mutual exclusion 117
 - DSM 113
 - first-in-first-out (PRAM) 114
 - leader/follower model 116
 - master-slave model 114
 - producer-consumer problem 115
 - release consistency model 117
- coroutines**
 - about 22
 - versus goroutines 21
- Couchbase**
 - about 216
 - data store, setting up 219, 220
 - features 216-218
- CouchDB**
 - about 214, 215
 - features 214
 - HTTP header syntax, using 214
- cpart.c 106**
- cpuprofile flag 144**
- CSP**
 - about 57
 - and actor model, difference between 60
 - dining philosophers problem 58, 59
- currentTime variable 99**
- customRouter**
 - serveStatic(): read function 166

D

- daemontools 240**
- data**
 - locking, mutex used 63-65
 - locking, sync package used 63-65
- data types**
 - sending, via channels 125
- deadlocks**
 - handling 150
- debugging**
 - LiteIDE, used for 248
- defer control mechanism**
 - Go's scheduler, using 15-20
 - implementing 13, 14
 - system variables, using 20, 21
- DELAY_INCREMENT value 273**
- DELETE syntax 214**
- Dial function 141**
- dining philosophers problem 58, 59**
- direct messages**
 - handling 131-134
- Distributed Go**
 - about 111, 112, 192
 - topology types 192
- Distributed Shared Memory. See DSM**
- Done() command 56**
- draw2d**
 - about 44
 - URL 44
- driver libraries, MySQL**
 - Go-MySQL-Driver 171
 - MyMySQL 171
- DSM 113**

E

- efficiency**
 - applying, in complex concurrency 67, 68
- endpoint patterns**
 - entrypoint/register/{name} 84
 - entrypoint/schedule/{name}/{time} 84
 - entrypoint/viewusers 84
- entrypoint/register/{name} 84**
- entrypoint/schedule/{name}/{time} 84**
- entrypoint/viewusers 84**

error handling
about 244-246
errors, logging to file 250, 251
errors, logging to memory 252, 253
errors, sending to screen 249

errors
handling 150
logging, to file 250, 251
logging, to memory 252, 253
sending, to screen 249

evalMessageRecipient function 134

evaluateStatus function 38

Execute() method 170

external dependencies

MySQL, connecting 171-173

F

file changes, concurrent application

detecting 224-226
records, checking against
Couchbase 229, 230
sending, to clients 226-229

file listener, concurrent application 206

files

errors, logging to 250, 251
working with 101, 102

files, concurrent application

backing up 231, 232

filesystem changes, concurrent application

monitoring 221

first-in-first-out (PRAM) 114

fsnotify 221

functionality, Go

panicking 259, 260
recovering 260-262

function channel

creating 126

G

garbage collection

used, in Go 203

gcfg

URL 223

generateHash() function 238

getArticle function 178

getFileDetails() method 262

getLetters function 33

Get method 57

GET syntax 214

Ginkgo

about 296

URL 296

gnuplot

URL 161

Go

assembly, using 109-111

blocking methods, using 124

concurrency impact, on I/O pprof 188-190

drawback 109

garbage collection, using 203

Ginkgo 296

GoCheck 296

Gomega 296

maps, using 128

NoSQL, using 209

parallelism impact, on I/O pprof 188-190

performance 182

polymorphism, demonstrating in 61, 62

pprof tool 143-150, 182-187

stack traces, catching with concurrent
code 265

structs, using 129

used, with unit testing 295

go:engine

about 44

URL 44

GoCheck

about 296

URL 296

Go' circuit

about 119, 120

URL 120

godoc

using 298

godoc command 298

GoFlow

about 202

URL 202

go get command 43, 50

GOMAXPROCS 62

Gomega 296

gomemcache interface

URL 118

Go-MySQL-Driver

URL 171

Google App Engine

packages 297

using 297

gopart.go 106

GOPATH 43

Gorilla toolkit

about 82

URL 82

goroutine logs

starting 246, 247

goroutines

about 9-11, 22

Blocking method 3 141

cleaning 29, 140

control, implementing with tomb 281-283

cost 100, 101

creating 33-35

used, for building web spider 35-39

versus coroutines 21

WaitGroup struct, implementing 11-13

working 41

go run command 202

GoSched() function 17

Go-SDL

about 44

URL 44

Go's scheduler

using 15-20

gosvg 42

grabFeed() method 52

graphics-go

about 44

URL 44

Graphviz

URL 146

H

happenedRlock() method 99

healthCheck function 289

Heka

about 202

URL 202

Hoare

about 57

URL 57

http.FileServer method 163

HTTP header syntax

DELETE 214

GET 214

PUT 214

http.ServeFile function 166

http template package 83

I

immutability 94, 95

indeterminate channel type

using 293

Initiate() function 133

InService property 288

interface channel

using 127

interfaces

using 128, 129

interval function 99

InUseBytes() method 203

InUseObjects() method 203

I/O pprof

concurrency, impacting on 188-190

parallelism, impacting on 188-190

itemsHandler function 53

ItemWrapper struct 234

J

JMeter

using 90

L

lastChecked variable 235

leader/follower model 116

letter capitalization, channels

example 25-28

libraries, Go

about 201

GoFlow 202

Heka 202

Nitro profiler 201

- `lines.SetRotateDaily()` 257
- `listen()` function 235
- listening loop, producer messages 279
- `Listen()` method 133, 134
- LiteIDE**
 - about 248
 - URL 249
 - used, for debugging 248
- load balancer**
 - building, with concurrent patterns 286-292
- load balancing** 273
- `Lock()` function 93
- `Lock()` method 99
- locks**
 - performing, on `RWMutex` 99
- log4go application**
 - about 222
 - URL 222
- log4go package**
 - URL 254
 - used, for robust logging 254-259
- logfiles, concurrent application**
 - managing 222
- logging**
 - about 244-246
 - benefit 263
 - goroutine logs, starting 246, 247
 - LiteIDE, used for debugging 248
- `log.SetRotateSize()` 257

M

- `main()` function 54, 144, 236, 283
- `main()` method 136
- `makefile` 106-108
- maps**
 - used, in Go 128
- master-slave model** 114
- memcached**
 - Go' circuit 119, 120
 - using 118, 119
- memory**
 - errors, logging to 252, 253
- memory leak**
 - creating 182
- memory preservation**
 - in Go 202

- mesh topology** 196, 197
- Message Passing Interface.** *See* **MPI**
- Message struct** 226, 234
- mgo**
 - URL 210
- MongoDB**
 - about 209-211
 - features 209
- MPI**
 - about 200
 - URL 201
- MPICH**
 - URL 201
- multiple cores**
 - leveraging 174
 - multithreading 174
- multiuser appointment calendar**
 - about 79, 80
 - concurrent pattern, visualizing 81
 - endpoints 84
 - example 85, 90-93
 - server requirements, developing 82
 - structs 85
- Mustache**
 - advantage 83
 - URL 83
- mutex**
 - used, to lock data 63-65
 - using 97-99
- mutex variable** 77
- mutual exclusions**
 - using 72-77
- MyMySQL**
 - URL 171
- MySQL**
 - connecting to 171-173
 - driver libraries 171

N

- `net.Listen()` method 130
- net package**
 - about 130
 - Client 130, 131
 - direct messages, handling 131-137
 - Server 130
- network connections** 141

nextServerIndex variable 290

nil channel blocks

implementing 278, 279

nil channels, using 280, 281

nil channels

using 280, 281

Nitro profiler

about 201

URL 201

NoSQL

URL 209

using 208

using, in Go 209

notFound() method 255

NumGcoCall() 111

O

object orientation, Go 60, 61

Open() method 172

OpenMPI

URL 201

os.OpenFile() method 250

P

panic() function 259, 260

panicking, Go 259, 260

panics

logging 263, 265

parallelism

impacting, on I/O pprof 188-190

Pipelined RAM (PRAM) 114

Platform-As-A-Service (paas) 297

polymorphism

demonstrating, in Go 61, 62

pprof tool 143-150, 181-187

private 93

producer-consumer problem 115

profiling tools 149

pseudocode, actor model 60

pseudocode, CSP 60

public 93

Publish and Subscribe model 197

PUT syntax 214

Q

Query command 178

QueryRow() function 178

R

race conditions

identifying, with race detection 68-72

mutual exclusions, using 72-77

timeouts, exploring 77, 78

race detection

race conditions, identifying with 68-72

Radix 211

re2 syntax

URL 134

reads 141

readURLs function 36, 39

receive-only channel

using 293

recover() function 259, 260

recovering, Go 260-262

Redigo 211

Redis

about 211

features 211

libraries 211

Redigo, using 211

URL 212

regexp.QuoteMeta() method 132

registerError() 265

release consistency model 117

remote code execution option

about 199

advantages 199

disadvantages 199

removeFile function 227

requests

handling 162-165

routing 165, 166

requisites, concurrent application

identifying 207, 208

restricted() function 256

Revel

URL 82

ReverseProxy method 290

- ReverseProxy struct** 291
- Rich Site Summary / Really Simple Syndication.** *See* **RSS**
- ring topology** 200
- robust logging**
 - log4go package, used for 254-259
- roundRobin function** 290
- RSS**
 - about 49
 - timeout, imposing 57
- RSS feeds**
 - parsing 50-53
 - retrieving 54-56
- RSS reader**
 - building 49
- runtime.Caller() function** 265, 267
- runtime.LockOSThread() function** 111
- runtime.Lookup function** 190
- runtime.MemProfileRecord object** 203
- runtime.NumGoroutine() function** 265, 268
- runtime package**
 - URL 268
 - used, for stack traces 265-268
- runtime.Stack() function** 265
- runtime.WriteTo method** 190
- RWMutex struct** 72, 98

S

- saveLogs() function** 267
- screen**
 - errors, sending to 249
- SELECT command** 178
- select statement**
 - using 29-33, 139
- send-only channel**
 - using 293
- separate packages**
 - building 299
- serial channel**
 - listening 124
- serialized data**
 - transmitting 198
- ServeHTTP method** 166
- server**
 - about 130
 - creating 273

- server health, concurrent application**
 - checking 241
- server requirements**
 - developing 82
 - templates, using 83, 84
 - time 84
 - web server 82
- servers**
 - failing, at concurrent connections 155, 156
- SetFormat option** 257
- setProxy function** 291
- SetRotateLines() method** 257
- SetRotate option** 257
- Sleep() method** 99
- Stack() method** 203
- stack traces**
 - catching, with concurrent code 265
 - runtime package, used for 265-268
- star topology** 193-195
- static pages**
 - external dependencies 171
 - serving 166, 167
 - template, parsing 167-170
- strings.TrimRight() method** 130
- structs**
 - about 85
 - used, in Go 129
 - using 128, 129
- style** 93
- SVG**
 - need for 44
- synchronous goroutine**
 - versus asynchronous goroutine 42
- sync.mutex struct** 98
- sync package**
 - about 97, 98
 - used, to lock data 63-65
- system variables**
 - using 20, 21

T

- template.ParseFiles() method** 170
- templates**
 - parsing 167-170
 - using 83, 84
- text template package** 83

- threads**
 - managing 62
- Tiedot**
 - about 213, 214
 - features 213
 - URL 214
- timeout**
 - exploring 77, 78
 - imposing 57
- time package 84**
- time.Ticker struct 284**
- tomb**
 - about 281
 - used, to implement control over goroutines 281-284
- tomb package**
 - URL 284
- tomb.Tomb struct 282**
- topN command 146**
- topology types**
 - bus topology 199
 - mesh topology 196, 197
 - MPI 200
 - Publish and Subscribe model 197
 - remote code execution option 199
 - ring topology 200
 - serialized data, transmitting 198
 - star topology 193-195
- transaction() function 75**
- Transport struct**
 - URL 57

U

- unbuffered channel 29**
- unidirectional channel**
 - selecting 292, 293
- unit testing**
 - Go, using with 295
- unlocks**
 - performing, on RWMutex 99
- unsafe package 185**
- updateFile function 229, 235**
- Upstart service, Ubuntu 240**
- URI variable 282**
- URL struct 282**
- url variable 51**

V

- visualization example 158, 159**

W

- Waitgroup struct**
 - implementing 11-13
- web-CLI interface, concurrent application 206**
- web command 146**
- Web.Go**
 - URL 82
- web interface, concurrent application**
 - designing 233-238
- web server**
 - about 82
 - connection mitigation 178, 179
 - exploring 174-178
 - history 155
- web server plan**
 - designing 42-44
- web spider**
 - building, channels used 35-39
 - building, goroutines used 35-39
- workers**
 - about 272
 - building 272-277



Thank you for buying Mastering Concurrency in Go

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

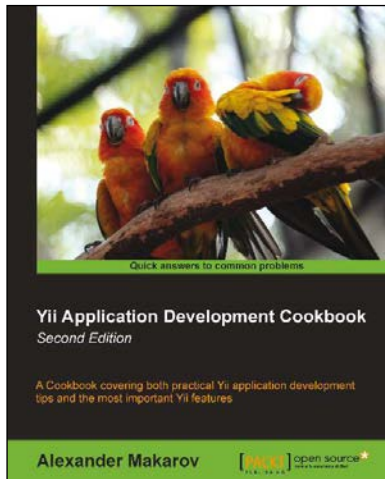
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Yii Application Development Cookbook Second Edition

ISBN: 978-1-78216-310-7

Paperback: 408 pages

A Cookbook covering both practical Yii application development tips and the most important Yii features

1. Learn how to use Yii even more efficiently.
2. Full of practically useful solutions and concepts you can use in your application.
3. Both important Yii concept descriptions and practical recipes are inside.



Real-time Web Application Development using Vert.x 2.0

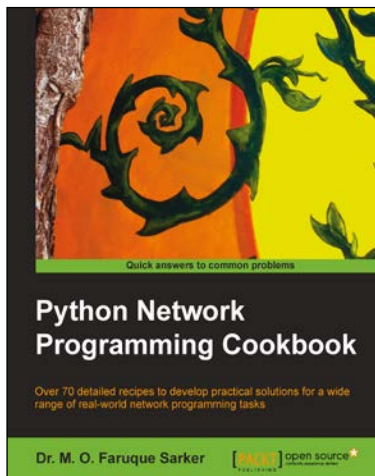
ISBN: 978-1-78216-795-2

Paperback: 122 pages

An intuitive guide to building applications for the real-time web with the Vert.x platform

1. Get started with developing applications for the real-time Web.
2. From concept to deployment, learn the full development workflow of a real-time web application.
3. Utilize the Java skills you already have while stepping up to the next level.

Please check www.PacktPub.com for information on our titles



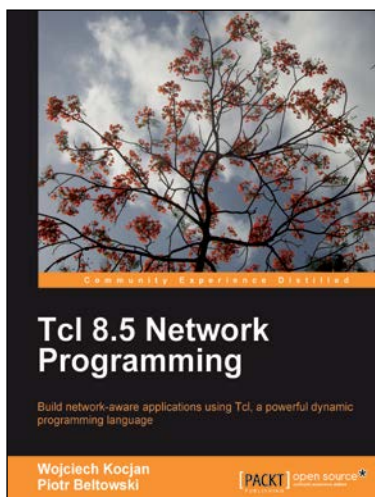
Python Network Programming Cookbook

ISBN: 978-1-84951-346-3

Paperback: 234 pages

Over 70 detailed recipes to develop practical solutions for a wide range of real-world network programming tasks

1. Demonstrates how to write various bespoke client/server networking applications using standard and popular third-party Python libraries.
2. Learn how to develop client programs for networking protocols such as HTTP/HTTPS, SMTP, POP3, FTP, CGI, XML-RPC, SOAP, and REST.
3. Provides practical, hands-on recipes combined with short and concise explanations on code snippets.



Tcl 8.5 Network Programming

ISBN: 978-1-84951-096-7

Paperback: 588 pages

Build network-aware applications using Tcl, a powerful dynamic programming language

1. Develop network-aware applications with Tcl.
2. Implement the most important network protocols in Tcl.
3. Packed with hands-on-examples, case studies, and clear explanations for better understanding.

Please check www.PacktPub.com for information on our titles