

# Building Services in Go

for fun and... profit?

**Hello!** If you want to code along, please make sure your Go environment is set up.

Either grab a USB stick that's floating around and copy the `gostick` directory and read the README, **or**,

```
git clone https://github.com/zorkian/lca2015.git
```

# Autobiography

- Mark Smith <mark@qq.is> @zorkian
- Site Reliability Engineer at Dropbox
- Contributes to Dreamwidth Studios
- [github.com/zorkian](https://github.com/zorkian)

# Today's Plan

- What is a service anyway?
- Why consider Go?
- Today's tutorial program
- Wrap-up

# Caveat Lector

- This is mostly not an "intro to Go" tutorial, neither is it advanced/guru level
- There will be a lot of code on slides, but we'll be talking through it
- Feel free to ask questions

# SERVICES



# SERVICES

## SERVICES

# SERVICES

SERVICES SERVICES

# Evolution of Design

- Let's talk briefly about philosophy
- Older sites are often large monolithic apps (Dropbox was/is, Dreamwidth is)
- Hard to extend and maintain — very complex systems over time, filled with dark corners



# Service Oriented Architecture

- Modern best practice seems to be very close to the Unix philosophy
- Small well scoped services that are easy to reason about
- Do something and do it well

# Why SOA?

- Largely, a way of managing complexity (reduces internal complexity)
- Narrow specs are easier to build and test
- Easier to monitor, and reasoning about failure is a much more straightforward proposition

# SOA

- Has been compared to building a highway system
- Works great with commodity hardware stacks
- Heavy use of the network

# Possible Drawbacks

- Sometimes simple tools are tough to combine in exactly the way you want — can lead to hacks
- Network calls do incur some overhead
- Maintaining 20 services has some more external complexity to maintain versus 1 big one



# Examples from Dropbox

- File torrent distributor that handles seeding files and nothing else
- Metadata service that is responsible for the structure of the logical filesystem
- Etc, etc



# Examples from Open Source

- nginx/varnish/haproxy
- memcached/redis
- ...etc

# Defining a Service

- Software that makes or receives requests, typically via an RPC layer
- Usually services communicate over the network

# So, why Go?

- Extremely well suited for building network services
- Language support for concurrency primitives
- Library support for RPCs and modern protocols

# Go :)

- Of course, all the other goodies that make it a great language to use (IMHO, YMMV, etc)
- Static typing (with interfaces for flexibility), static and fast compilation, simple/easy to learn, Unicode by default, ...

# Tutorial time!

- Yay! So you want to build a service?
- Today we're going to build a pretty commonly useful kind of thing I love — a proxy :)
- In specific, a simple (but relatively performant) HTTP proxy



# Why a proxy?

- Everything can be solved with a proxy
- Great for gathering statistics or changing behavior of opaque systems (think databases!)
- Good combination example of client/server behavior, with good illustration of Go tenets

# Basic Design

- Accept a connection
- Read some requests
- Proxy to backend
- Write response to client

# Follow Along

- Open the file `gostick/part1/main.go`
- You can write code as we go along if you want
- Alternately, open `gostick/part1_final/main.go` and visually follow along

# I. Listening

```
func main() {  
  
    // 1. Listen for connections forever.  
    if ln, err := net.Listen("tcp", ":8080"); err == nil {  
        for {  
  
            // 2. Accept connections.  
            if conn, err := ln.Accept(); err == nil {  
                // ...more code...  
            }  
        }  
    }  
}
```

This looks like any other language...  
Yes, we're ignoring errors.



# Go net/http Library

- The standard library has great support for network servers
- Full suite of HTTP code to make a basic proxy extremely trivial
- Makes heavy use of bufio...



# What is bufio?

- Many small reads/writes is very inefficient
- If you ever see high system CPU%, you might be inefficiently doing I/O
- Use bufio for performance: read like normal, or write (and remember to flush!)

## 2. Reading a Request

```
func ReadRequest(b *bufio.Reader) (req *Request,  
                                err error)
```

```
    if conn, err := ln.Accept(); err == nil {
```

```
}
```

## 2. Reading a Request

```
func ReadRequest(b *bufio.Reader) (req *Request,  
                                err error)
```

```
    if conn, err := ln.Accept(); err == nil {
```

```
        reader := bufio.NewReader(conn)
```

```
}
```

## 2. Reading a Request

```
func ReadRequest(b *bufio.Reader) (req *Request,  
                                err error)
```

```
    if conn, err := ln.Accept(); err == nil {  
        reader := bufio.NewReader(conn)  
  
        // 3. Read requests from the client.  
  
        if req, err := http.ReadRequest(reader); err == nil {  
  
        }  
    }  
}
```

# 3. Talking to Backends

```
if be, err := net.Dial("tcp", "127.0.0.1:8081"); err == nil {
    be_reader := bufio.NewReader(be)
}
```

Again, pretty simple. Note another bufio.



## 4. Proxying Request to BE

```
if err := req.Write(be); err == nil {  
    // 6. Read the response from the backend.  
  
    if resp, err := http.ReadResponse(be_reader, req); err == nil {  
        // ... doing something ...  
    }  
}
```

The net/http library makes our job pretty easy.

# 5. Send Response to Client

```
if resp, err := http.ReadResponse(be_reader, req); err == nil {  
    // 7. Send the response to the client.  
    resp.Close = true  
    if err := resp.Write(conn); err == nil {  
        log.Printf("%s: %d", req.URL.Path, resp.StatusCode)  
    }  
    conn.Close()  
    // Repeat back at 2: accept the next connection.  
}
```

For this dumb proxy, we're not doing keep-alive.

# Let's Test...

- If you followed along, you should have a program that is hideously nested and ignores all errors
- But it should work ;)

# Building Part I

```
# Do this in your gostick/ copy
```

```
# You did follow the README? :-)
```

```
cd part1/          # or part1_final
```

```
go build
```

```
./part1            # or part1_final
```

# The Backend

```
# Do this in a new terminal and
```

```
# then forget about it
```

```
cd webserver/
```

```
go build
```

```
./webserver
```



# Test!

- In your browser, try:  
`http://127.0.0.1:8080/`
- You should see the Go documentation, and in your terminal, a couple lines of output
- Shout if you have issues

# Part 2

- That example was quite... lacking
- Serialized everything
- One customer at a time
- Overall, just really slow (benchmarks at ~500qps)

# Going Faster

- Need to start to think in some form of “doing two things at once”: async, parallel, concurrent, etc
- In Go, it’s all about concurrency, and it’s a central feature of the language

# Concurrency

- In essence, write everything to be blocking
- You spin off many blocking operations into concurrently running goroutines
- The runtime is responsible for scheduling in new goroutines when you block



# Proxy Concurrency

- Logically, each client talking to the proxy is going to be operating separately and independently
- Seems like a good candidate for having one goroutine per connection
- Go is happy with 100,000+ goroutines!



# Rewriting main()

```
func main() {  
    ln, err := net.Listen("tcp", ":8080")  
    if err != nil {  
        log.Fatalf("Failed to listen: %s", err)  
    }  
    for {  
        if conn, err := ln.Accept(); err == nil {  
  
        }  
    }  
}
```

Open part2/main.go! Add some error handling.

# Rewriting main()

```
func main() {  
    ln, err := net.Listen("tcp", ":8080")  
    if err != nil {  
        log.Fatalf("Failed to listen: %s", err)  
    }  
    for {  
        if conn, err := ln.Accept(); err == nil {  
            go handleConnection(conn)  
        }  
    }  
}
```

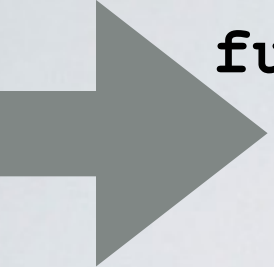
Add a call to a new function. Note the **go** keyword!

# handleConnection()

```
func handleConnection(conn net.Conn) {  
    defer conn.Close()  
    reader := bufio.NewReader(conn)  
  
    for {  
        req, err := http.ReadRequest(reader)  
        if err != nil {  
            if err != io.EOF {  
                log.Printf("Failed to read request: %s", err)  
            }  
            return  
        }  
        // more code will go here  
    }  
}
```

Handles one incoming connection, closes it when this routine exits. Has error handling!

# handleConnection()

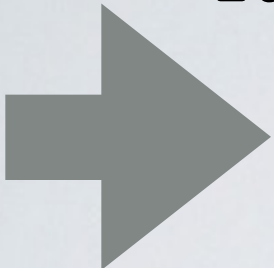


```
func handleConnection(conn net.Conn) {  
    defer conn.Close()  
    reader := bufio.NewReader(conn)  
  
    for {  
        req, err := http.ReadRequest(reader)  
        if err != nil {  
            if err != io.EOF {  
                log.Printf("Failed to read request: %s", err)  
            }  
            return  
        }  
        // more code will go here  
    }  
}
```

Handles one incoming connection, closes it when this routine exits. Has error handling!



# handleConnection()

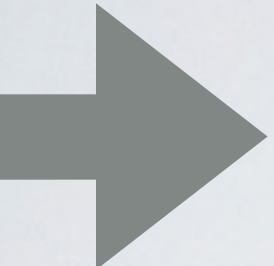


```
func handleConnection(conn net.Conn) {  
    defer conn.Close()  
    reader := bufio.NewReader(conn)  
  
    for {  
        req, err := http.ReadRequest(reader)  
        if err != nil {  
            if err != io.EOF {  
                log.Printf("Failed to read request: %s", err)  
            }  
            return  
        }  
        // more code will go here  
    }  
}
```

Handles one incoming connection, closes it when this routine exits. Has error handling!



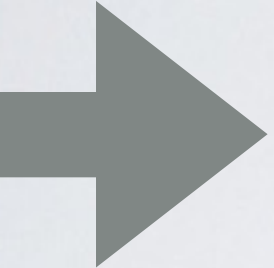
# handleConnection()



```
func handleConnection(conn net.Conn) {  
    defer conn.Close()  
    reader := bufio.NewReader(conn)  
  
    for {  
        req, err := http.ReadRequest(reader)  
        if err != nil {  
            if err != io.EOF {  
                log.Printf("Failed to read request: %s", err)  
            }  
            return  
        }  
        // more code will go here  
    }  
}
```

Handles one incoming connection, closes it when this routine exits. Has error handling!

# handleConnection()



```
func handleConnection(conn net.Conn) {  
    defer conn.Close()  
    reader := bufio.NewReader(conn)  
  
    for {  
        req, err := http.ReadRequest(reader)  
        if err != nil {  
            if err != io.EOF {  
                log.Printf("Failed to read request: %s", err)  
            }  
            return  
        }  
        // more code will go here  
    }  
}
```

Handles one incoming connection, closes it when this routine exits. Has error handling!

```

// Connect to a backend and send the request along.
if be, err := net.Dial("tcp", "127.0.0.1:8081"); err == nil {
    be_reader := bufio.NewReader(be)
    if err := req.Write(be); err == nil {
        if resp, err := http.ReadResponse(be_reader, req); err == nil {
            if err := resp.Write(conn); err == nil {
                log.Printf("%s: %d", req.URL.Path, resp.StatusCode)
            }
        }
    }
}

```

Same code from part I — just move it down!

# Building Part 2

```
# Can test and build now, make
```

```
# sure webserver is still up!
```

```
cd part2/          # or part2_final
```

```
go build
```

```
./part2            # or part2_final
```



# It's faster...

- The new version benchmarks at about 2000 qps
- Definitely faster, but we're still re-establishing outbound connections every request
- We're also not actually doing anything useful



# Part 3

- Let's gather some statistics about requests
- This is going to be pretty simple... but you could extend it

# Statistics!

```
var requestBytes map[string]int64
var requestLock sync.Mutex

func init() {
    requestBytes = make(map[string]int64)
}
```

We're going to count the total bytes on each path and include that data in the response headers.

The data structure has to be initialized.

Note we're now using **sync.Mutex**!

# updateStats()

```
func updateStats(req *http.Request, resp *http.Response) int64 {  
    requestLock.Lock()  
    defer requestLock.Unlock()  
  
    bytes := requestBytes[req.URL.Path] + resp.ContentLength  
    requestBytes[req.URL.Path] = bytes  
    return bytes  
}
```

Small, testable functions! Also note defer usage again, we have to lock the global structure for safety.

We're depending on empty-is-0 from the map.

# Er, a mutex in Go?

- Go encourages you to “share memory by communicating”, so we could
- Channels are extremely good for cross-thread coordination (passing ownership, workers, etc)
- Mutexes are good for... well, you know



# Mutex vs Channel

- Use whichever construct is the most simple way to get your job done
- Channels will often be the best way to express complicated patterns
- In our simple global state case... mutex :)



# Back to our code...

```
if resp, err := http.ReadResponse(be_reader, req); err == nil {  
    bytes := updateStats(req, resp)  
    resp.Header.Set("X-Bytes", strconv.FormatInt(bytes, 10))  
  
    if err := resp.Write(conn); err == nil {  
...  
}
```

We have to call our new function, and then we want to get the result and put it into our proxy response header.

The magic of proxies! We could have done something fun to the resulting web page, but this is enough for now.

# Building Part 3

```
# Can test and build now, make
```

```
# sure webserver is still up!
```

```
cd part3/          # or part3_final
```

```
go build
```

```
./part3            # or part3_final
```

# Testing Stats

- To see your code in action, you should try to test with curl:

```
curl -v -o /dev/null \  
    http://127.0.0.1:8080/
```

# Part 4

- Let's fix the backend to pre-generate and use connection pooling
- Also, this work solves the ephemeral port issue (which mostly happens during benchmarking)
- Code starting point as usual in `part4/main.go`



# Queues in Go

```
type Backend struct {  
    net.Conn  
    Reader *bufio.Reader  
    Writer *bufio.Writer  
}
```

Note the embedded type!

```
var backendQueue chan *Backend  
var requestBytes map[string]int64  
var requestLock sync.Mutex
```

```
func init() {  
    requestBytes = make(map[string]int64)  
    backendQueue = make(chan *Backend, 10)  
}
```

Buffered channel!



# Backend Manufacturing

```
func getBackend() (*Backend, error) {  
    select {  
        case be := <-backendQueue:  
            return be, nil  
        case <-time.After(100 * time.Millisecond):  
            be, err := net.Dial("tcp", "127.0.0.1:8081")  
            if err != nil {  
                return nil, err  
            }  
  
            return &Backend{  
                Conn: be,  
                Reader: bufio.NewReader(be),  
                Writer: bufio.NewWriter(be),  
            }, nil  
    }  
}
```

Standard Go style with error return. This constructs the buffered I/O objects for us, since we always want to use them!

# Backend Manufacturing

```
func getBackend() (*Backend, error) {  
    select {  
        case be := <-backendQueue:  
            return be, nil  
        case <-time.After(100 * time.Millisecond):  
            be, err := net.Dial("tcp", "127.0.0.1:8081")  
            if err != nil {  
                return nil, err  
            }  
  
            return &Backend{  
                Conn: be,  
                Reader: bufio.NewReader(be),  
                Writer: bufio.NewWriter(be),  
            }, nil  
    }  
}
```

Standard Go style with error return. This constructs the buffered I/O objects for us, since we always want to use them!

# Backend Enqueueing

When done with backends, we have to get rid of them.

```
func queueBackend(be *Backend) {  
    select {  
        case backendQueue <- be:  
            // Backend re-enqueued safely, move on.  
        case <-time.After(1 * time.Second):  
            be.Close()  
        }  
    }
```

We want to try to keep the backends around, but we don't want to block forever. Discards if the timer fires.

# Updating handleConnection()

```
if be, err := net.Dial("tcp", "127.0.0.1:8081"); err == nil {  
    be_reader := bufio.NewReader(be)  
    if err := req.Write(be); err == nil {  
        if resp, err := http.ReadResponse(be_reader, req); err == nil {
```

Old code above, new code below. Uses backendQueue channel, which is thread-safe automatically!

```
be, err := getBackend()  
if err != nil {  
    return  
}  
  
if err := req.Write(be.Writer); err == nil {  
    be.Writer.Flush()  
    ...  
}  
go queueBackend(be)
```



# Building Part 4

```
# Can test and build now, make
```

```
# sure webserver is still up!
```

```
cd part4/          # or part4_final
```

```
go build
```

```
./part4            # or part4_final
```



# What's next?

- Still one place where a bufio needs using (writing responses back to clients)
- If we have a traffic lull, the backends in the queue can get old and become invalid
- Maybe keep the queue warm by early connects

# More Next

- Can add a lot more statistics and stuff
- Structure of the program is not very extensible
- Check out the `final/` directory for my original implementation (which is 2.5x faster than part4)

# Part5 - Adding RPCs

- We're probably low on time, so let's look quickly at doing RPCs in Go
- Built in RPC library, serialization, etc, makes it really easy to build

# Server Implementation

```
type Empty struct{  
type Stats struct {  
    RequestBytes map[string]int64  
}  
type RpcServer struct{  
  
func (r *RpcServer) GetStats(args *Empty, reply *Stats) error {  
    requestLock.Lock()  
    defer requestLock.Unlock()  
  
    reply.RequestBytes = make(map[string]int64)  
    for k, v := range requestBytes {  
        reply.RequestBytes[k] = v  
    }  
    return nil  
}
```



# Server Side (2)

```
func main() {  
    rpc.Register(&RpcServer{})  
    rpc.HandleHTTP()  
    l, err := net.Listen("tcp", ":8079")  
    if err != nil {  
        log.Fatalf("Failed to listen: %s", err)  
    }  
    go http.Serve(l, nil)  
  
    // ...  
}
```



# Client Side

```
func main() {  
    client, err := rpc.DialHTTP("tcp", "127.0.0.1:8079")  
    if err != nil {  
        log.Fatalf("Failed to dial: %s", err)  
    }  
  
    var reply Stats  
    err = client.Call("RpcServer.GetStats", &Empty{}, &reply)  
    if err != nil {  
        log.Fatalf("Failed to GetStats: %s", err)  
    }  
  
    // use reply.RequestBytes
```

# Whew!

- We've covered a lot of ground today, but it's not time for questions yet
- Let's talk some gotchas!

# Go is Still New

- Missing some things people really want, like generics... some code is annoying to write
- Best practices are still being ironed out

# Garbage Collection

- In high volume services, remember that Go has a GC that presently has to pause the world
- Keep memory allocations in check, and use free pools if you're doing a lot

# Libraries

- The standard libraries are pretty good, but the ecosystem is still only a few years old
- Third party libraries are a huge mishmash: some are great, some are not, some are still supported, some are not...
- Dropbox libs at [github.com/dropbox/godropbox](https://github.com/dropbox/godropbox)



# GOMAXPROCS

- Generally you want to set GOMAXPROCS to some value  $> 1$  for high volume services
- Benchmarking is required! There are breakpoints in performance (depends on your app and how much you use channels, etc)

# In short...

- Best practices from C/Java/etc are mostly the same
- Go doesn't mean you can just forget about everything forever, it just pushes it off to later
- ...so maybe you never have to worry!

# Thank you!

Mark Smith <mark@qq.is> @zorkian  
SRE at Dropbox (we're hiring!)

- Some recommended reading:  
**Effective Go** [golang.org/doc/effective\\_go.html](http://golang.org/doc/effective_go.html)  
**FAQ** [golang.org/doc/faq](http://golang.org/doc/faq)
- Many more talks and presentations:  
[code.google.com/p/go-wiki/wiki/GoTalks](http://code.google.com/p/go-wiki/wiki/GoTalks)
- The Go Playground! [play.golang.org](http://play.golang.org)