Overview

1. Building proxy using Go
   - Part I
   - Part 2
   - Part 3
   - Part 4

Proxy

- You are going to build a proxy service
- In specific, a simple (but relatively performant) HTTP proxy

Why 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 *dspdemocode/go/GolangTraining/src/proxy/part1/main.go*
- You can write code as we go along if you want
- Alternately, open *dspdemocode/go/GolangTraining/src/proxy/part1final/main.go* and visually follow along

1. 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 {

   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

Building Part 1

```
cd part1/ # or part1_final
go build
./part1 # or part1_final
```

The Backend

```
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

Part 2

- That example was quite slow
- 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!

handleConnection()

```
// 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 part1 — just move it down!

Building Part 2

```
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

- 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.

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)

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

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

```
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/
```

- Let's fix the backend to pre-generate and use connection pooling
- Code starting point as usual in part4/main.go

Queues in Go

Note the embedded type!

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

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 Enqueueing

```
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)
```

62

Building Part 4

```
cd part4/ # or part4_final
go build
./part4 # or part4_final
```

Part5 - Adding RPCs

- 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
```