Go Tutorial and Lab 1  
Overview

Joe DeBlasio
CSE 223B, Spring 2015
Slides graciously from He ‘Lonnie’ Liu
TA office hours

EBU3B B275
● Monday 1-2am
● Wednesday 10-11am
Lab 1 Due

Thu 4/9, 11:19am
Does everyone have a lab account?

- 22 of you haven’t changed your password yet
- 15 of you haven’t logged into the lab machines first
- 10 machines
  - vm143.sysnet.ucsd.edu
  - ...
  - vm152.sysnet.ucsd.edu
- SSH only

- Need a lab account? Tell me by 2pm today
Coding on your own machine

- Install git, go
- Setup ssh
- Clone the repos
Install Go

Linux: download and untar to /usr/local/go

OSX (Mac): download and install
GOROOT & GOPATH

GOROOT=/usr/local/go

GOPATH=$HOME/gopath

- Could have multiple paths
- sub-dirs: src/, pkg/, bin/
Go tools

go get/install/build/run/doc/fmt/...

Example:
    go get github.com/jstemmer/gotags

Formatting source code:
    go fmt
Vim and Tags

```
filetype off
filetype plugin indent off
set runtimepath+==$GOROOT/misc/vim
filetype plugin indent on

gotags -R . > tags
```
Lab Details
Update the triblab repo, please!

$ cd ~/gopath/src/triblab
$ git fetch
$ git merge --no-edit origin/master
Hanging Tests

Don’t forget to send true to Ready
Handling Empty List

- Problem: `ListGet()`
- RPC marshall/unmarshall will optimize away nil/empty lists.
- In `rpc.Call("Storage.ListGet")`, fetching an empty list will leave the list parameter untouched.
- Solution: set `ret.L` to an empty list before the call.
client.Close()

- Textbooks say we should always **Close()**
- **Close()** is needed when committing updates
  - e.g. writing to a file
- **Close()** is needed for releasing resources
  - (e.g. opening a socket).
- **RPC** is committed when the function returns.
- **Persistent connections** do not take more resources.
- **Close()** is hence optional.
- OS will eventually clean up everything.
Persistent Connection

- So… you’ve been tempted to optimize..
- It improves performance but makes the design much more complex.
- The performance requirement is easy to satisfy.
  - Round trip time, around 1ms
  - TCP open, around 1ms
- Thus, it’s just an optimization.
  - Optimization is the source of evil.
  - So, just don’t do it.
  - No really, you’ll just lose points.
Persistent Connection

- But you still want to do it…
- Save the connection, of course.
- **Challenge**: How to tell if the connection is closed already?
- **Trick**: after the connection is closed (with error or not), all subsequent calls will return `rpc.ErrShutdown`
- **Strategy**: if return `rpc.ErrShutdown`, the call is always not performed, so reconnect and try again.
Starting the back-end

- Don’t use the global server (DefaultServer). Create your own.
- `http.Serve(listener, handler)` takes two arguments, both are interfaces.
  - First is `net.Listener`, should be `TCPLListener`, can create from `net` package.
Golang Basics
Consts

cost n = 3
const i = n + 0.3
const N = 3e9
const (  
    bitA = 1 << iota  
    bitB  
    bitC
)
cost s = "a string"
Variables

```go
var i = 0
var i int = 0
var ( 
    i = 0
) 
var i 
i := 0
```
Runes and Strings

```go
def run(a rune = 'a'
def run(a rune = '囧' // utf-8 coding point
def s string = "a\n\t"
def s2 string = `multi-line
string`
```
Types

type D struct {}
type D int
type D struct { a int }
type D struct { next *D }
**Functions**

```go
def main() {
}
def (d *D) Do() {
}
def (d *D) Write() (n int, e error) {
}
def (d *D) private() {
}
```
Array and slices

```go
var a [3]int
var b, c []int
b = a[:]
c = b[:]
c = b

// a = b[:] // error
b = a[2:] // to the end
b = a[:3] // from the start
println(len(a)); println(cap(a))
```
Append

```go
var a []int
a = append(a, 2)
a = append(a, 3, 4, 5)
var b []int = []int{6, 7, 8}
a = append(a, b...)
```
Maps

m := map[string]int {
    "one": 1, "two": 2, "three": 3,
}

m["four"] = 4
delete(m, "four")
i := m["four"]
i, found := m["four"]
For

for i := 0; i < 3; i++ { }
for i < 3 { } // like while
for { } // infinite loop
for index := range slice { }
for index, element := range slice { }
for key := range map { }
for key, value := range map { }
for index, rune := range str { }
Switch

switch a {
    case 0:
        // no need to break
    case 2:
        fallthrough
    default:
}
Switch (2)

switch {
    case a < 2:
    case a > 10:
    default:
}

Defer

```go
func (s *server) get() {
    s.mutex.lock()
    defer s.mutex.unlock()
    _get() // perform the action
}
```
Interfaces

type D struct {}
type Writer interface {
    Write()
}

func (d *D) Write() {}
var _ Writer = new(D)
Anonymous fields

type D struct {}
func (d *D) Get()
var d *D = new(D)
d.Get()

type E struct { *D }
var e *E = &E{d}
e.Get()
e.D.Get()
Channel

c := make(chan int) // cap(c)=0

c := make(chan int, 3) // cap(c)=3

var in chan<- int = c

in <- 2; in <- 3

var in <- chan int = c

a := <- c
Go routine

go f()

time.Sleep(time.Second)
runtime.Gosched() // yield
runtime.Goexit() // exit
Select

select {
    case <-c1:
    case <-c2:
    case <-timer:
    default:
}

Commonly used packages

os // Stdin, Stdout
io, io/ioutil // Reader, Writer, EOF
bufio // Scanner
fmt // Print[ln], Printf, Fprintf, Sprintf
strings // HasPrefix/Suffix, Fields, Trim
bytes // Buffer
time // Time, Duration
net // TCPConn, UDPCConn, IPCConn
sort // Interface
More packages

encoding/json, encoding/binary
math, math/rand
hash/fnv
net/http
sync
log, debug
path, path/filepath
flag
container/heap(,list,ring)