Go Tutorial

Anil Yelam
ayelam@eng.ucsd.edu
CSE 223B, Spring 2020
Administrative details

TA Office Hours:
  Tuesday 10-10:45 AM
  Thursday 2:30-3:15 PM
  Zoom links on Canvas

TA Email: ayelam@eng.ucsd.edu
Administrative details

All labs due by 2359 PDT.
• Lab 1 due: 4/09/2020.
• Lab 3 due: 5/07/2020.

Lab 1 is easy; Labs 2,3 are much harder. Start early.
Outline

• Labs setup (15 mins)

• Go Basics (30 mins)

• Some advice on Labs (15 mins)
Labs overview

• Labs 1-3 will work with a Twitter-like service called Tribbler.

• We provide a non-fault-tolerant, non-scalable single machine implementation of Tribbler.

• You will apply what you learn in the class to make Tribbler fault-tolerant and scalable.
What you will *not* be doing

• Students will *not* change the basic Tribbler service.

• Specifically, nothing in the ‘trib’ directory (details later) will be modified—but you’re welcome to read it.

• Instead, we will write code exclusively in the ‘triblab’ repository.
Infrastructure: EC2 VMs

UCSD Account: awsed.ucsd.edu

- 50$ AWS credits
- Pre-configured Image
- Instructions here

You can use your personal account too. Free-tier suffices!
Infrastructure: EC2 VMs

Recommended. Why?

• I’ll be able to answer your questions promptly

• You might need a couple more VMs for testing in later labs
Coding on your own system
(unsupported!)

Caveat: no support offered.
Code must compile and run without any modification to tests/scripts.

Remember that I will be grading on EC2 VMs with original scripts. If you need to change scripts, let’s talk!
Setting up

• Use assignment invitation link

• Follow instructions in the Repo
Go tools

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

Example:  
go get github.com/jstemmer/gotags

Formatting source code:  
go fmt
Golang Basics
What is Go?

Go is:
• An imperative programming language.
• ...that is fully garbage collected.
• ...which has pointers, but *no* pointer arithmetic.
• ...that has interfaces and structs (that might implement interfaces), but no objects or inheritance.
Why use Go?

• Go has a variety of features that aid: multithreaded and asynchronous programming and making RPC calls.

• This will make our life easier when writing distributed systems.

• There’s also a bunch of other useful features we’ll go over.
Types in Go

- Basic types (signed and unsigned integers of various sizes, booleans, runes, strings, pointers to various things...)

- Arrays and slices

- Maps

- Channels

- Interfaces

- Structs
Basic types: numerical

- Signed and unsigned integers exist, either of specific size or not

- Eg. int, int8, int64, float32, float64, complex128

- No automatic casting! Even if the underlying representation is the same!

- A string is a collection of arbitrary bytes

- false and true are bools
Variable declarations

```plaintext
var i = 0
var i int = 0
var (   
    i = 0
   )
var i
i := 0
```
Const declarations

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

```go
var a rune = 'a'
var a rune = '囧' // utf-8 coding point
var s string = "a\n\t"
var s2 string = `multi-line
string`
```
Basic types: pointers

• Points to an object

• Similar to C/C++ in that sense.

• But: no pointer arithmetic.

• In golang, function parameters are ‘pass by value’ so pointers can be used if we want the method to modify the input parameter.
Getting pointers to things.

• We can get the address of an object with the ‘&’ operator.

• We can return the address of a locally created variable and have it be valid after the method returns (unlike C/C++).
Interfaces

• An interface is a collection of methods.

• An interface has a name, so do all the methods.

• If a type implements all these methods, it implements the interface.

• An interface can contain other interfaces.
Structs

- A struct is a type sort of like in C.
- It can aggregate multiple types inside it.
- It can implement an interface.
- All we need to do is write an implementation of every method in the interface, for the struct.
Functions

• Go has ‘first class functions’

• We can pass functions as arguments, we can get functions as results.

• Go also has closures: a way to encapsulate the environment that a function was created in.

• Functions can return multiple values.

• For more info, look online or ask a programming languages person.
Function definition

```go
func Sum(a, b int) (int, error) {
    return a + b, nil
}
```

- 'func' means it’s a function
- 'Sum' is the name
- 'a' and 'b' are parameters, they’re both int
- Sum returns an ‘int’
- The body is within curly braces
- This function is just a single addition.
Functions vs. methods

• A Method is a function that is associated with a particular type.

• The definition is almost like a function, except there is one additional parameter: the ‘receiver’.

• The receiver is like the ‘this’ pointer in C++.

• Methods can help a type implement an interface.
Method example

- type MyInteger int

  func (this MyInteger) MyMethod(b int) int {
    return this + b
  }

- Here we define MyInteger as an int.

- We make a method MyMethod that operates on a MyInteger called ‘this’ (like this in C++).

- The rest should be familiar.
Functions

```go
func main() { }
func (d *D) Do() { }
func (d *D) Write() (n int, e error) { }
func (d *D) private() { }
```
Interfaces

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

func (d *D) Write() { }
var _ Writer = new(D)
Memory management and creating objects

- Garbage collected: we create but don’t need to delete things

- There’s a heap and stack like in C/C++ but we don’t need to worry about the details

- We create things with: new, make and initializer lists
Creating objects

- new creates a 0-initialized object and returns a pointer to it.

- make is used to create slices, maps and channels only (more on that in a bit) but returns an object, not a pointer.

- An initializer list allows us to create a struct with certain values for each member, or an array or map initialized with certain initial values.
Data structures: Maps

• Not a lot to say here: it’s like maps in other languages.

• Should be familiar if you know python dicts or c++ maps.

• Note: not concurrent access safe. Use a mutex to access if there’s multiple threads using a map.
Maps

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

m["four"] = 4
delete(m, "four")
i := m["four"]
i, found := m["four"]
Data structures: Arrays and Slices

• An Array is a fixed size, fixed type array.

• An integer array of size 3 is NOT the same type as one of size 4.

• Instead: just use slices.

• Slices are created via make() and have syntax similar to python lists.

• Alternatively, you can initialize a slice with values.
Data structures: Arrays and Slices

• Slices refer to some subset of the underlying array (can refer to the whole array).

• Recommendation: just use slices.
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...)
```
Control flow

• **If** is used for evaluating conditionals.

• **For** is used for looping.

• **Switch** is also available; note that by default, switch cases don’t fallthrough (unless you call fallthrough).

• There’s no ternary if operator.
For

```plaintext
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:
    }
More Control flow

- **Defer** is used to schedule a function to be called *after* the current function is done.

- Multiple defer executed in LIFO fashion.

- **Panic** and **Recover** are for when very bad errors occur; you probably won’t be using it. They’re not like C++ exceptions.

- **Instead**: use C style error handling.
Defer

func (s *server) get() {
    s.mutex.lock()
    defer s.mutex.unlock()
    _get() // perform the action
}
Data structures: Channels

• Channels are conduits that can be used to communicate between threads.

• You can send any type of object over a channel, including channels.

• Think of them as really useful pipes in Unix.
Data structures: Channels

• Channels can be unbuffered or buffered.

• An unbuffered channel means that a writer to a channel will block till a reader processes the object written to the channel: WATCHOUT FOR DEADLOCK!

• A buffered channel of size N means we can write up to N objects before the channel is full (after which the writer blocks).

• We can use select on the read side to poll channels.
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
Select

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

Goroutines and Multithreading

- **Goroutines** can be used to execute a function in its own thread.

- **Channels** can be used to communicate data between threads.

- We can also use shared memory with mutexes like in C/C++.

- Goroutines are multiplexed on underlying OS threads.
Go routine

go f()

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

• We can define methods in such a way that they can be remotely exposed

• There’s an input param, an output param, and the whole function returns an Error (which is nil if it succeeded)

• Go can expose 1 instance of an object type over RPC only.

• Eg. If there are cats and dogs as types, we can expose 1 cat and 1 dog, but not 2 dogs.
Types

type D struct {}
type D int
type D struct { a int }
type D struct { next *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()
Logging

import "log"

log.Print(object)
log.Panic(object)
log.Fatal(object)
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, UDPConn, IPConn
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)
Lab 1 specific advice
Hanging Tests?

- You’ll notice that some of the methods involve sending ‘true’ to a channel when your code is ready.

- This tells the test code that it can proceed with the tests.

- So: don’t forget to send true to the ‘Ready’ channel when the assignment calls for it!
Handling Empty Lists

The default go RPC encoding (‘gob’) has trouble telling nil and empty lists apart!

Example:
var someList = new(trib.List)
someList.L = []string{"item1", "item2"}
log.Printf("Length of list: %v", len(someList.L))
ret := rpc.ListGet("some_empty_key", &someList}
log.Printf("Length of list: %v", len(someList.L))

This should output:
2
0

But it outputs:
2
2
Handling Empty Lists

Two possible solutions:

1. set someList.L to nil before the call, replace someList.L with an empty list after the call if someList.L is nil.

2. Use JSON encoding instead of GOB.
Error registering multiple RPC endpoints

“Default RPC path was already registered”
Or
All backends point to same storage object

One solution:
Use different ServeMuxes instead of the default one. See net/http for more.
Questions

Try going through the Lab descriptions again before posting questions.

More important for later labs!
Next Labs

Lab 1 is a warmup for GO. Labs 2 & 3 are more distributed systems.

Read through the assumptions, requirements, building hints and possible mistakes.
Questions on Piazza
AI violations

• Ok to search for golang code/repos.

• Just don't go searching for things about the labs in particular!
‘make upstream’ bug

[FYI] Minor issue with Lab 1 Makefile

There's a bug in the Makefile that links upstream repository (make upstream) improperly and fails to make update suggest you to pull the fix from upstream as bug prevents exactly that.

So if you see permission issues with make update, please manually remove the upstream repo and re-add it properly. After that

```bash
git remote remove upstream
git remote add upstream https://github.com/ucsd-cse223b-sp20/lab1-starter
make update
```
Cloning starter code

- Clone your repo now!
Other questions?
Good luck!

• And start early!!