More Go, Lab 1 Hints, and MapReduce

Tom Anderson

Last Time

- MapReduce
  - Computational model
  - Implementation
  - (not yet: fault tolerance)
  - (not yet: discussion)
- Remote Procedure Call
  - Introduction

Topics

- Go
  - Synchronization
  - RPC
- MapReduce
  - Fault tolerance
  - Discussion
- RPC
  - At least once, exactly once, at most once

Go Unicode

Remember that a string is NOT an array of one byte characters.

Any particular character can be variable size, so you need to use the appropriate library code for parsing strings.

Go Functions as Data

```
// define a function
f := func(c rune) bool {
	return !unicode.IsLetter(c)
}

// type of f
var f func(rune) bool

// can then pass f to a function, e.g., FieldsFunc takes f
// to determine where its safe to split words
tokens := strings.FieldsFunc(value, f)
```

Go Libraries

```
// all the docs are online; google is your friend
import "strings"

func FieldsFunc(s string, f func(rune) bool) []string
```

FieldsFunc splits the string s at each run of Unicode code points c satisfying f(c) and returns an array of slices of s.

Go Anonymous Functions

```
// this is the same as:  fmt.Println("Hello World!")
func(){
  fmt.Println("Hello World!")
}()

// we can pass arguments to anonymous functions
func(n uint){
  for (i:= 0; i < n; i++) {
    fmt.Println("Hello World!")
  }
}(5)
```

Go Threads

Can create a thread to run a function (named or anonymous) in the background

```
go func(){
  fmt.Println("Hello World!")
}()
```

Note: program exits when main thread exits, even if some threads are still running

Go Channels

```
// channel = typed bounded buffer
toDo = make(chan int, 10)

// put work in
toDo <- 1

// take work out
i:= <- toDo

// Implemented as a slice with locks and condition variables
```

Select and Channels

```
// To wait for multiple things, use select as an event loop
// wait for one of the channels to have work.
for {
	select {
	case address := <- mr.registerChannel:
	// a new worker is registering
	case nextTask := <- mr.mapToDo:
	// there's a map task to do
	case nextTask := <- mr.reduceToDo:
	// there's a reduce task to do
}}
```

Select and Channels

```
// you can also add conditions to select operations
for {
	select {
	case address := <- mr.registerChannel:
	// a new worker is registering
	case worker && nextTask := <- mr.mapToDo:
	// there's a map task to do
	case worker && nextTask := <- mr.reduceToDo:
	// there's a reduce task to do
}}
```

Channels and Concurrency

Can use channels to create a computational pipeline:

- op1 | op2 | op3
- Two channels, three or more threads

Go and CSP

Can use channels to replace shared objects

- create a thread to manage the object
- create a channel for incoming requests
- create a channel for outgoing replies
- thread loops waiting for work to come in on the incoming channel
  - does operation and puts result on outgoing channel
- can have multiple input channels if multiple types of work to do (using select)

CSP vs. Monitors

Monitors:

- Set of threads, that acquire a lock before calling into an object, so that only one thread executes inside the monitor at a time
- Use condition variables to wait until ok to do some op

CSP:

- one thread executes all the operations on the object
- other threads invoke object methods by sending object a message (on a channel)
- Use select to wait until ok to do some op

Lab 1 Notes

Part 1: write a simple MapReduce program to do word count. Master runs the code worker code directly.

Part 2: write the master, where workers run as separate processes

Files:

common.go -- the RPC spec, shared between client and server
mapreduce.go -- code for splitting the initial file into chunks, creating file names, etc.
master.go -- code for managing workers
worker.go

Go RPC Conventions

By convention, all RPC's have two arguments and return an error code:

- Funcname(arg *FuncArgs, reply *FuncReply) error

By convention, all RPC's have function names that are capitalized

- the system takes all of those as RPC's, whether you intended them or not

If you get errors like this, just ignore them:

- go test
- 2012/12/28 14:51:47 method Kill has wrong number of ins: 1

Lab 1 Notes

For mapreduce, master and workers: which is the client and which is the server?

Lab 1 RPCs: worker -> master

// Workers initialize, then register with the master

func (mr *MapReduce) Register(args *RegisterArgs, res *RegisterReply) error

Lab 1 RPCs: master->worker

// Need an RPC to do map/reduce task on worker

func (wk *Worker) DoJob(arg *DoJobArgs, res *DoJobReply) error

// And an RPC to tell worker to shut down

func (wk *Worker) Shutdown(args *ShutdownArgs, res *ShutdownReply) error

RPC Errors

ok := call(worker, "Worker.DoJob", args, &reply)

// if ok is true, call performed
// if ok is not true, was call performed?

// For Lab 1, if ok = false, worker did not and will never touch the intermediate files that it was asked to create
// For Lab 2, need to handle the more general case

RPCs and Concurrency

- RPCs are blocking on the client
  - MapReduce master needs multiple RPCs to be outstanding – why?
- Need a thread per worker or a thread per RPC
- Keep track of which tasks are still to be done, and which are complete.
- Only start Reduce tasks once all Map tasks are done
- If you hand a Map task to a worker, and the RPC fails, then you need to hand it to a different worker -- e.g., put it back on the task list

RPCs and Concurrency

- RPCs are concurrent on the server
  - In Lab 1, only one master, so (hopefully!) not an issue
  - In Lab 2+, need to use locks or channels to protect any shared data
    - Do not mix styles
  - With locks, be careful to use good locking hygiene!
    - Lock at beginning of RPC
    - Release at end of RPC

Is an RPC like a normal function call?

Binding
- Client needs a connection to server
- Server must implement the required function
- What if the server is running a different version of the code?

Performance
- local call: maybe 10 cycles = ~3 ns
- in data center: 10 microseconds => ~1K slower
- in the wide area: millions of times slower

Failures
- What happens if messages get dropped?
- What if client crashes?
- What if server crashes?
- What if server appears to crash but is slow?
- What if network partitions?

MapReduce Fault Tolerance Model

Master is not fault tolerant
- Assumption: this single machine won't fail during running a mapreduce app

Many workers, so have to handle their failures
- Assumption: workers are fail stop
- They can fail and stop
- They may reboot
- They don't send garbled weird packets after a failure

What kinds of faults does MapReduce need to tolerate?

- Network:
- Worker:

Tools for Dealing With Faults

- Retry
  - if pkt is lost: resend
  - worker crash: give task to another worker
  - may execute MR job twice! (is this ok? Why?)
- Replicate
  - E.g., input files
- Replace
  - E.g., new worker can be added

Lab 1 MapReduce Simplifications

- No key in map
- Assume global file system
- No partial failures
  - Files either completely written or not created
  - If restart some failed operation, ok to write to the same filename

DeWitt/Stonebraker Critique

- A giant step backward in the programming paradigm for large-scale data intensive applications
- A sub-optimal implementation, in that it uses brute force instead of indexing
- Not novel at all: represents a specific implementation of well known techniques developed nearly 25 years ago
- Missing most of the features that are routinely included in current DBMS
- Incompatible with all of the tools DBMS users have come to depend on