SparkGeo: Assignment

Submitted on August 5, 2015

Will Cadell

Mehul Solanki

Contents

List of Figures

List of Tables

Abstract

This document consists of the final submission according to the requirements and also the thinking process by it was reached. It also lists the resources used and code sections which may not necessarily be a part of the final submission.

Requirements

Let's start by getting the requirements which were mentioned in the email. A small snipped is shown below,

> ....
> *Have you worked with GO before? Here is our little test:*
> *We would like a piece of GO code written that will receive http requests (lots of them) and add them to a worker queue, to be processed subsequently, in order.*
> *In essence this piece of code is like a catcher of http requests that ensures a processing framework doesnt get overloaded by the requests by using a queue instead of setting off processes directly. For the purposes of this project you can keep all processes generic (ie they dont have to actually do anything). Does this make sense?*
> *Ideally you could put this code onto an amazon instance :)*
> ....

The first thing that popped out to me was GO Lang. I had played around with it some time ago so I decided to go gather the documentation [1, 5] and video content [14, 15].

Environment Details and Setup

1. Operating System : Kubuntu 14.04 64 bit, 3.13.0-45-generic kernel.
2. Hardware : 12 Core Intel i7-4960X @ 3.60 GHz.
3. Memory : 27.4 GiB RAM
4. GO Lang : go version go1.4.2 linux/amd64
5. Development Environment : Sublime Text 3 - Build 3083, Plugins - Go, Go Tools, Go Sublime, Go Gdb, Go Oracle Results
6. Terminal : Konsole Version 2.13.2

Documenting

Sr No.	Technology	Resource	Notes
1	TeXStudio 2.6.6	[17]	Latex Editor
2	Minted	[12]	Source Code Highlighter
3	Tex Live 2013	[11]	pdflatex with -shell-escape flag

Table 1: Documenting Technologies

Queue development

Since the requirements stated something about a queue, I decided to build / use a queue and try and play around with a bit by populating it with HTTP requests along with reading basics in Go lang. Searching through the documentation I was not able to find a library for abstract data structures though there was an example for priority queues. I found a simple queue implementation [13] and staring playing with it.

```go
package queueds
/*Queue in go*/
import (
        "fmt"
)
type Node struct {
        Value int
}
func (n *Node) String() string {
        return fmt.Sprint(n.Value)
}
func NewQueue(size int) *Queue {
        return &Queue{
                nodes: make([]*Node, size),
                size:  size,
        }
}
// Queue is a basic FIFO queue based on a circular list
//that resizes as needed.
type Queue struct {
        nodes []*Node
        size  int
        head  int
        tail  int
        count int
}
// Continued on next page
```

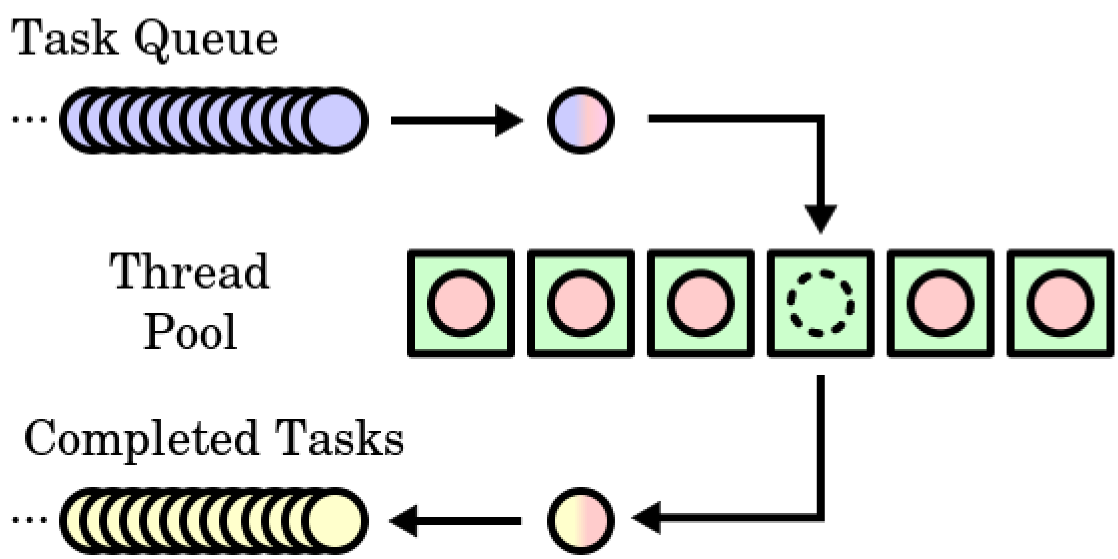
```
// Push adds a node to the queue.
func (q *Queue) Push(n *Node) {
        if q.head == q.tail && q.count > 0 {
                nodes := make([]*Node, len(q.nodes)+q.size)
                copy(nodes, q.nodes[q.head:])
                copy(nodes[len(q.nodes)-q.head:],
                        q.nodes[:q.head])
                q.head = 0
                q.tail = len(q.nodes)
                q.nodes = nodes
        }
        q.nodes[q.tail] = n
        q.tail = (q.tail + 1) % len(q.nodes)
        q.count++
}
// Pop removes and returns a node from the queue in first
//to last order.
func (q *Queue) Pop() *Node {
        if q.count == 0 {
                return nil
        }
        node := q.nodes[q.head]
        q.head = (q.head + 1) % len(q.nodes)
        q.count--
        return node
}
func main() {
        q := NewQueue(1)
        q.Push(&Node{4})
        q.Push(&Node{5})
        q.Push(&Node{6})
        fmt.Println(q.Pop(), q.Pop(), q.Pop())
}
```

There were also few other things that I found such as Task Queues [7] and Heaps [6] but felt that something simpler might come up. Also, developing a small chink of code would help in understanding.

Concurrency in Go Lang

Chapter 10 of [1] talks about go routines and channels which is the main starting point of the implementation. The primary resources for understanding about worker queues and worker-crew models are [19, 16].

Figure 1: Worker-crew Model



The figure above from [19] provides details about the Thread Pool Pattern. A number a threads are created to perform a number of tasks, the latter outnumbering the former quite a bit. The results can be stored in a separate queue. So the threads(workers) will request for a job; upon finishing it request for another and so on. A dispatching mechanism will send work to the workers.

Attempting something like this in Go Lang with the help of go-routines, delays and multiple bidirectional channels.

```go
// 1/5

package main

import(
        "fmt"
        "time"
        "math/rand"
)

/*
Making progress on more than one task simultaneously is known
as concurrency. Go has rich support for concurrency using
goroutines and channels.
*/


// GOROUTINE
// capable of running concurrently with other functions.
// keyword : go func_name function_invocation
// Normally when we invoke a function our program will
// execute all the statements in a function and then
// return to the next line following the invocation.
// With a goroutine we return immediately to the next
// line and don't wait for the function to complete.

func f(n int) {
for i := 0; i < 10; i++ {
//Printing
fmt.Println(n, ":", i)
// Genrating Random Time between 0 and 250
amt := time.Duration(rand.Intn(250))
// Converting number into ms to sleep
time.Sleep(time.Millisecond * amt)
}
}
// Continued on next page...
```

```
// 2/5

func f1(){

        for i := 0; i < 10; i++ {
// Creating multiple go routines
                go f(i)
}

}

// CHANNELS
// provide a way for two goroutines to communicate
// with one another and synchronize their execution.
// keyword : chan type_of_data_passed_on_chan
// send operator : <-, msg := <- c
// recieve : chan_name <- msg
// synchronizes the two goroutines.

func pinger(c chan string) { // Bi-directional
for i := 0; ; i++ {

// send / put message on channel
c <- "ping"

}

}

func printer(c chan string) { // Bi-directional
for {
msg := <- c // recieve msg from channel c
fmt.Println(msg)// print
time.Sleep(time.Second * 1)// wait time,
}
}
// Continued on next page ...
```

```
// 3/5

func ponger(c chan string) { // Bi-directional
for i := 0; ; i++ {
c <- "pong"
}
}

// Channel Direction : either sending or recieving

// Sender function
// func pingerOnly(c chan<- string)

// Reciever Function
// func printerOnly(c <-chan string)

func main() {// already a go routine, implicit

go f(0)// go routine
var input string

/*
Blocking
When pinger attempts to send a message on the channel it will
wait until printer is ready to receive the message.
*/
// Creating a new string channel
var c chan string = make(chan string)
go pinger(c)// Putting msg on channel c (Ping)
go ponger(c)// Putting msg on channel c (Pong)
go printer(c)// recieving message on channel c

// Select : just like switch but for channels
c1 := make(chan string)
c2 := make(chan string)

// Continued on next page ...
```

```
// 4/5

go func() {
for {
c1 <- "from 1"// send on c1
time.Sleep(time.Second * 2)//wait for 2 seconds
}
}()

// If more than one of the channels are ready then
// it randomly picks which one to receive from.
// If none of the channels are ready, the statement
// blocks until one becomes available.
go func() {
for {
c2 <- "from 2"// send on c2
time.Sleep(time.Second * 3)//wait for 3 seconds
}
}()

// Continued on next page ...
```

If more than one channels are ready then the go routine(continued on next page) will choose randomly since the decision is made by the select statement. If none are available then it blocks until one is available.

```
// 5/5

go func() {
for {
select {

//Case for c1
case msg1 := <- c1:
fmt.Println(msg1)

//Case for c2
case msg2 := <- c2
fmt.Println(msg2)

//Case for timeout
case <- time.After(time.Second):
fmt.Println("timeout")

//default case if nothing is ready
default:
fmt.Println("nothing ready")
}
}
}()

buffChan := make(chan int, 10)// chan type, chan capacity

// included for the wait so that the goroutine can finish
fmt.Scanln(&input)

// Printing input value
fmt.Println(input)

}
```

HTTP in Go Lang

The next step is to implement and use some HTTP functionality in a program and maybe catch a few requests here and there. Starting with the `Package HTTP` and some examples mentioned in the articles [8]. The steps include developing a handler function and then using `http.HandleFunc` and `http.ListenAndServe`.

```go
// (1/1)
package main
import (
        "fmt"
        "io"
        "net/http"
)
// Handler Function.
func hello(res http.ResponseWriter, req *http.Request) {
        res.Header().Set(
                "Content-Type",
                "text/html",
        )
        io.WriteString(
                res,
                `<DOCTYPE html>
<html>
  <head>
      <title>Hello World</title>
  </head>
  <body>
      Hello World!
  </body>
</html>`,
        )
// Prints every time the page is visited.
        fmt.Println("hello")
}
func main() {
        http.HandleFunc("/hello", hello)
        http.ListenAndServe(":9000", nil)
}
```

Catcher

Developing the example a little further to implement the first part of the program,

```go
// (1/3)
// Mehul Solanki
// solanki@unbc.ca, ms280690@gmail.com
// SparkGeo Test
package main
import (
        "fmt"
        "net/http"
        "strconv"
        "time"
)

// The catcher queue must be implemented through channels in go,
// since subroutines(catching and dispatching) can communicate
// and synchronize over it. But go channels need to be typed,
// so we need to define a structure of messages that the
// channel will work on.

// Channel Structure.
// We can add other fields later.
type RequestStructure struct{
// In the curl command the following data will be sent,
// The current user running the command.
 userName string
// To send multiple request, a for loop will be used so the
// value of the variable(i).
 requestNumber int
// The current unix timestamp in unix.
 requestTimeStamp string
}
// Creating new channel of type RequestStructure with
// buffer size 1000.
var catcherQueue = make(chan RequestStructure, 1000)
```

```
// (2/3)
// The required fields are extracted from the request and
// converted to the appropriate format.
func catcherFunction(rs http.ResponseWriter, rq *http.Request){
// For creating a work request we need a userName and
// requestNumber.
 userName := rq.FormValue("userName")
// Since the return type is string we need to convert
// it to an int.
 requestNumberString := rq.FormValue("requestNumber")
// Creating the RequestStructure to put on the channel.
 workRequest := RequestStructure{ userName: userName,
 requestNumber: requestNumber,
 requestTimeStamp: requestTimeStampString}

// Printing the request.
 fmt.Println("userName", userName)
 fmt.Println("requestNumber", requestNumberString)
 fmt.Println("requestTimeStamp", requestTimeStampString)

// Put it on the queue.
 catcherQueue <- workRequest
 fmt.Println("From Catcher : Work request queued")

// Creating a status.
 rs.WriteHeader(http.StatusCreated)
 fmt.Println("From Catcher : StatusCreated")
 fmt.Println()
 return
}
```

```go
// (3/3)
func main() {

        http.HandleFunc("/work", catcherFunction)
        http.ListenAndServe("127.0.0.1:1025", nil)
}
```

The first part implements the structure that the channel can handle. A `requestStructure` has three properties; *userName* : who initiated the request, *requestNumber* and *requestTimeStamp* : when was the request made. The `catcherQueue` is a buffered channel with size 1000.

Next, the handler function `catcherFunction`. Upon receiving a request(generated and sent by the *curl* command discussed later) it converts the input data into the required structure discussed above and puts it on the queue.

Lastly, putting it all together in a similar way mentioned in the example from the previous section.

Worker

```go
// (1/3)
package main
import (
        "fmt"
)

// Next is to create something which will handle the requests,
// a Worker.
type Worker struct {
// Worker id
 id int
// The channel where the work is received.
 work chan RequestStructure
// The channel where the workers reside.
 workerQueue chan chan RequestStructure
}
```

```go
// (2/3)
// Make function to create a new worker, just like the
// make function for other types such as channels.
func makeWorker(id int,
            workerQueue chan chan RequestStructure) Worker {

// Create, and return the worker.
        worker := Worker{
                id:         id,
// Each worker has it's own queue where the work is received.
                work:       make(chan RequestStructure),
// Common, where all the workers are.
                workerQueue: workerQueue,
        }

        return worker
}
```

```
// (3/3)
// This function initializes each worker.
// func_name() allows it to be called on a
// struct using the . operator
func (w Worker) initWorker() {

// This is to replicate the while(true) infinite loop structure.
 for {

// Put the worker in the worker queue.
        w.workerQueue <- w.work

// Switch case for channels
                select {

// Receive a work request.
                 case work := <- w.work:

// Who is working? // Whose work is it?
        fmt.Printf("From Worker %d : Work Request received,
                 at your service %s", w.id, work.userName)
        return
                        }
    }
}
```

The first part of the program describes the structure of the workers. So there is a queue(chan) *workerQueue* of workers and each one has their own queue(chan) *work*. The point to be noted here is that why is the *workerQueue* a `chan chan`? The logic is that the entity which sends a request is the real consumer which I learnt from [10].

The next part creates a worker while the last part is to initialize. Also I used [3] to replicate the `while(true)` condition / control-flow in Go Lang.

Dispatcher

```
// (1/1)
package main

import (
        "fmt"
)
// This function initializes the workers.
func initDispatcher(workerCount int) { // number of workers

// Initializing the common channel where the workers will wait
// for the work.
workerQueue := make(chan chan RequestStructure, workerCount)

// Depending on the count, creating the required
// number of workers.
        for i := 0; i < workerCount; i++ {
         fmt.Println("From Dispatcher : Starting Worker", i+1)
         worker := makeWorker(i+1, workerQueue)
// Since we want the workers to be running and
// ready to receive work requests simultaneously,
// so it has to be a go routine
         go worker.initWorker()
        }
```

Mentioned above is the code required to initialize the *workerQueue* depending on the number of workers. Simply speaking we create the queue which holds the work channels for each worker.

```
// (2/2)
// Outer Anonymous Go routine.
 go func() {
        for {
                select {
// Pull off work request from the queue
                 case work := <-catcherQueue:
                  fmt.Println("From Dispatcher :
                                  Received work requeust")
// Inner Anonymous Go routine.
                         go func() {

                                 worker := <-workerQueue
                                  fmt.Println("From Dispatcher :
                                   Dispatching work request")
// Send the work request to worker.
                                 worker <- work
                                 }()
                         }
                }
        }()
}
```

The outer *go routine* gets a request off the queue and the inner *go routine* sends it to the worker.

Main

```
// (1/1)
package main

import (
        "fmt"
        "net/http"
)

func main() {

// Start the dispatcher.
 fmt.Println("From Main : Starting the dispatcher")
 initDispatcher(10)

// Register our catcherFunction as an HTTP handler function.
 fmt.Println("From Main : Registering the catcher")
 http.HandleFunc("/work", catcherFunction)

// ListenAndServe HTTP server!
 fmt.Println("From Main : HTTP server listening on",
 "127.0.0.1:1026")

 err := http.ListenAndServe("127.0.0.1:1026", nil)

 if err != nil {
         fmt.Println(err.Error())
        }
}
```

The final task is to put everything together. This section is very similar to the ones mentioned previously Catcher and HTTP in Go Lang.

Running

Building

Since we want the "application" to run we have to build it like one and generate an object file [4], we use the following command

```
go build -o program *.go
```

Execution

And then run it,

```
./program
```

curl Command

In a separate terminal run,

```
for i in {1..20};
do curl localhost:1026/work
        -d userName=$USER
        -d requestNumber=$i
        -d requestTimeStamp=$(date "+%S:%M:%k//%d-%m-%y");
done
```

My main references are [2, 9, 18]. The data being sent here is in accordance to the *requestStructure* of our channels.

Output

Let the output flow,

```
From Main : Starting the dispatcher
From Dispatcher : Starting Worker 1
From Dispatcher : Starting Worker 2
From Dispatcher : Starting Worker 3
From Dispatcher : Starting Worker 4
From Dispatcher : Starting Worker 5
```

```
From Dispatcher : Starting Worker 6
From Dispatcher : Starting Worker 7
From Dispatcher : Starting Worker 8
From Dispatcher : Starting Worker 9
From Dispatcher : Starting Worker 10
From Main : Registering the catcher
From Main : HTTP server listening on 127.0.0.1:1026
userName root
requestNumber 1
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 1 : Work Request received, at your service root
userName root
requestNumber 2
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 2 : Work Request received, at your service root
userName root
requestNumber 3
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 3 : Work Request received, at your service root
userName root
requestNumber 4
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
```

```
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 4 : Work Request received, at your service root
userName root
requestNumber 5
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 5 : Work Request received, at your service root
userName root
requestNumber 6
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 6 : Work Request received, at your service root
userName root
requestNumber 7
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 7 : Work Request received, at your service root
userName root
requestNumber 8
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated
```

```
From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 8 : Work Request received, at your service root
userName root
requestNumber 9
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 9 : Work Request received, at your service root
userName root
requestNumber 10
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
From Dispatcher : Dispatching work request
From Worker 10 : Work Request received, at your service root
userName root
requestNumber 11
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
userName root
requestNumber 12
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
userName root
requestNumber 13
requestTimeStamp 36:11:16//05-08-15
```

```
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
userName root
requestNumber 14
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
userName root
requestNumber 15
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
userName root
requestNumber 16
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
userName root
requestNumber 17
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
userName root
requestNumber 18
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated
```

```
From Dispatcher : Received work requeust
userName root
requestNumber 19
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
userName root
requestNumber 20
requestTimeStamp 36:11:16//05-08-15
From Catcher : Work request queued
From Catcher : StatusCreated

From Dispatcher : Received work requeust
```

Bonus

Since the requirements stated that the amazon instance part was **"ideal"** I will send it later.

References

[1] Caleb Doxsey. *An Introduction to Programming in Go*. Google, 1st edition edition, 2012.

[2] LAKSHMANAN GANAPATHY. 15 practical linux curl command examples (curl download examples). http://www.thegeekstuff.com/2012/04/curl-examples/, 2012.

[3] Google. Forever. https://tour.golang.org/flowcontrol/4, 2015.

[4] Google. Go build. http://golang.org/src/cmd/go/build.go, 2015.

[5] Google. Go lang documentation. https://golang.org/doc/, 2015.

[6] Google. Go lang documentation. http://golang.org/pkg/container/heap/, 2015.

[7] Google. Task queue go api overview. https://cloud.google.com/appengine/docs/go/taskqueue/, 2015.

[8] Google. Writing web applications. https://golang.org/doc/articles/wiki/, 2015.

[9] Haxx. curl. http://curl.haxx.se/docs/manpage.html, 2015.

[10] Traun Leyden. Understanding chan chan's in go. http://tleyden.github.io/blog/2013/11/23/understanding-chan-chans-in-go/, 2013.

[11] TeX Live. Tex live. https://www.tug.org/texlive/, 2015.

[12] minted. minted. https://code.google.com/p/minted/, 2015.

[13] Rodrigo Moraes. Lifo stack and fifo queue in golang. https://gist.github.com/moraes/2141121, 2013.

[14] Rob Pike. Concurrency is not parallelism. https://www.youtube.com/watch?v=cN_DpYBzKso, 2012.

[15] Rob Pike. Go concurrency patterns. https://www.youtube.com/watch?v=f6kdp27TYZs, 2012.

[16] RabbitMQ. Work queues. https://www.rabbitmq.com/tutorials/tutorial-two-python.html, 2015.

[17] TeXstudio. Texstudio. http://www.texstudio.org/, 2015.

[18] Wikipedia. curl. https://en.wikipedia.org/wiki/CURL, 2015.

[19] Wikipedia. Thread pool pattern. https://en.wikipedia.org/wiki/Thread_pool_pattern, 2015.