Buzzer

Find out what everyone's buzzing about!

Taeber Rapczak
taeber@ufl.edu

COP5618 Concurrent Programming
Spring 2019

https://cise.ufl.edu/~trapczak/buzzer

#Overview

Buzzer is a microblogging service written in Go on which users socialize by posting messages known as *buzzes*.

Registered users can subscribe to another users posts which appear on their *buzz-feed* along with any message in which they were mentioned (@username).

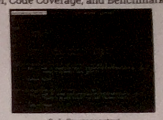
Users can search for messages by tags (#topic) or for other users.

Buzzer uses Go's channels and goroutines to coordinate the asynchronous activity and expose the service via a WebSockets-based API for real-time, bidirectional communication with a web client.

The web client is written using HTML5 (including the WebSocket API), Facebook's React JavaScript framework, and uses some of the responsive-design elements of CSS3. It should be usable from any modern smartphone.

#Tools

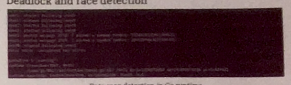
Test Runner, Code Coverage, and Benchmarking



Code Coverage output

```
$ make benchmark
GOPATH=/home/taeber/code/cop5618-concurrent/project/lib:/home/taeber/code/cop5618-concurrent/project go test -bench=. -run=^$ buzzer -benchmem
goos: linux
goarch: amd64
pkg: buzzer
BenchmarkKernelPost-4          200000      4415 ns/op     657 B/op       9 allocs/op
BenchmarkChannelServerPost-4   200000      6450 ns/op     634 B/op       9 allocs/op
PASS
ok      buzzer  2.744s
```

Deadlock and race detection



Data race detection in Go runtime

#Objectives

Primary

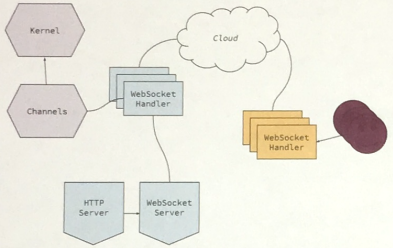
Use Go's goroutines and channels to handle concurrency issues learned from COP5618 Concurrent Programming. Specifically, how to:

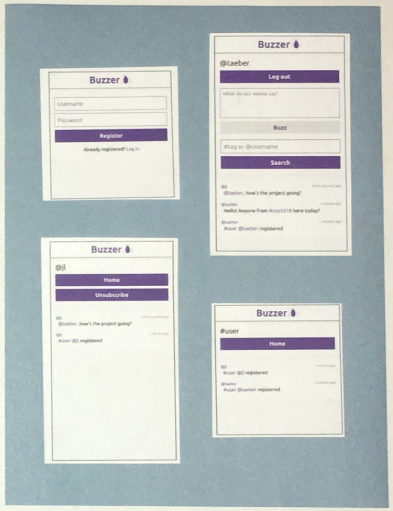
- provide mutual exclusion or atomic updates to parts of code;
- utilize processor resources when code blocks;
- minimize hazards through safe publication and immutability.

Secondary

Write a non-trivial project in Go and report on the aspects of the languages and tools that were helpful in developing the software.

#Design





#Conclusions

Goroutines are the agents.

Goroutines, Channels, and `select` provide an alternative to traditional shared memory concurrency concepts.

It is still easy to have concurrency issues/bugs when using channels.

The Go runtime and tools provides useful detection of data races and issues arising from non-determinate behavior.

Go's build tools provide convenient tooling to development software.

Go language constructs and library calls are used by the runtime to suspend Goroutines as needed.

#Background

Goroutine

Lightweight thread managed by the Go runtime.

Execute concurrently with other functions.

Channel

Conduits to send and receive messages, which are typed values.

Message sent **happens-before** message received.

Sending and receiving is a synchronous operation.

Buffered channels allow for multiple messages to be sent or received before blocking.

`select` statement operates on multiple channels. `default` case makes communication non-blocking.

WebSockets

TCP-based protocol for bidirectional communication over ports 80 and 443 (for Transport Layer Security).

#Locking

```
func newChannelServer(actual *kernel) *channelServer {
    // Use channels to sync access to actual kernel.
    return &channelServer{
        actual:   actual,
        post:     make(chan request, 100),
        follow:   make(chan request, 100),
        unfollow: make(chan request, 100),
        messages: make(chan request, 100),
...
        login:    make(chan request, 100),
        logout:   make(chan request, 100),
        shutdown: make(chan bool),
    }
}

func (server *channelServer) process() {
    for {
        select {
        case req := <-server.post:
            msgID, err := server.actual.Post(req)
            go respond(&req, response{data: msgID, error: err})
...
        case <-server.shutdown:
            return
        }
    }
}
```

#ConditionalSync

```
go func() { // Inside accept()
  defer func() { shutdown <- true }()
  for { // Processes any messages received.
    select { // Guarded suspension.
    case msg := <-received:
      client.decodeAndExecute(msg)
    case sub := <-client.subscribe:
      username := client.getUsername()
      if sub.follower != username {
        continue
      }
      if sub.unfollow {
        client.Write("unfollow "+sub.followee)
      } else {
        client.Write("follow " + sub.followee)
      }
    }
  }
}()
<-shutdown // Wait for shutdown.
```

#Semaphore

```
client := wsClient{
    username:  make(chan string, 1),
    …
}

// client.username is used as a semaphore.
go func() {
    client.username <- ""
}()

func (client *wsClient) getUsername()
(username string) {
    username = <-client.username
    client.username <- username
    return
}

func (client *wsClient) setUsername(username
string) {
    <-client.username
    client.username <- username
}
```

#FutureWork

Implement the backend using more traditional shared memory primitives like locks and conditional variables to compare performance.

Write an Actor Model framework for Go or a language based on the Go library and runtime.

#Thanks

A Tour of Go, Golang.org Authors.
https://tour.golang.org/concurrency
WebSocket Protocol, Fette and Melnikov.
https://tools.ietf.org/html/rfc6455
Preparing and Presenting Effective Research Posters, Miller, Jane.
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1955747/

GO React

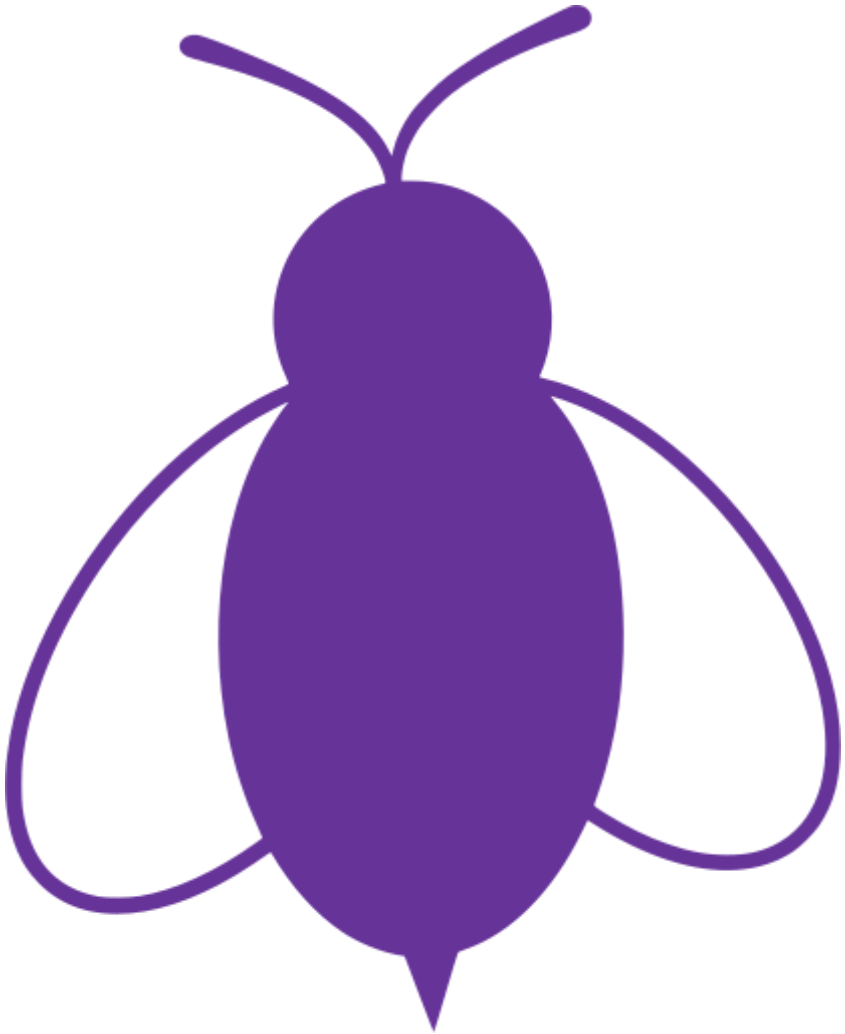
UF UNIVERSITY of FLORIDA

Buzzer

Find out what everyone's buzzing about!

Taeber Rapczak
taeber@ufl.edu

COP5618 Concurrent Programming
Spring 2019



https://cise.ufl.edu/~trapczak/buzzer

#Overview

Buzzer is a microblogging service written in Go on which users socialize by posting messages known as *buzzes*.

Registered users can subscribe to another users posts which appear on their *buzz-feed* along with any message in which they were mentioned (@username).

Users can search for messages by tags (#topic) or for other users.

Buzzer uses Go's channels and goroutines to coordinate the asynchronous activity and expose the service via a WebSockets-based API for real-time, bidirectional communication with a web client.

The web client is written using HTML5 (including the WebSocket API), Facebook's React JavaScript framework, and uses some of the responsive-design elements of CSS3. It should be usable from any modern smartphone.

#Objectives

Primary

Use Go's goroutines and channels to handle concurrency issues learned from COP5618 Concurrent Programming. Specifically, how to:

- provide mutual exclusion and atomic updates to parts of code;
- utilize processor resources when code blocks;
- minimize hazards by communicating instead of sharing memory.

Secondary

Write a non-trivial project in Go and report on the aspects of the languages and tools that were helpful in developing the software.

#Background

Goroutine

Lightweight thread managed by the Go runtime.

Execute concurrently with other functions.

Channel

Conduits to send and receive messages, which are typed values.

Message sent **happens-before** message received.

Sending and receiving is a synchronous operation.

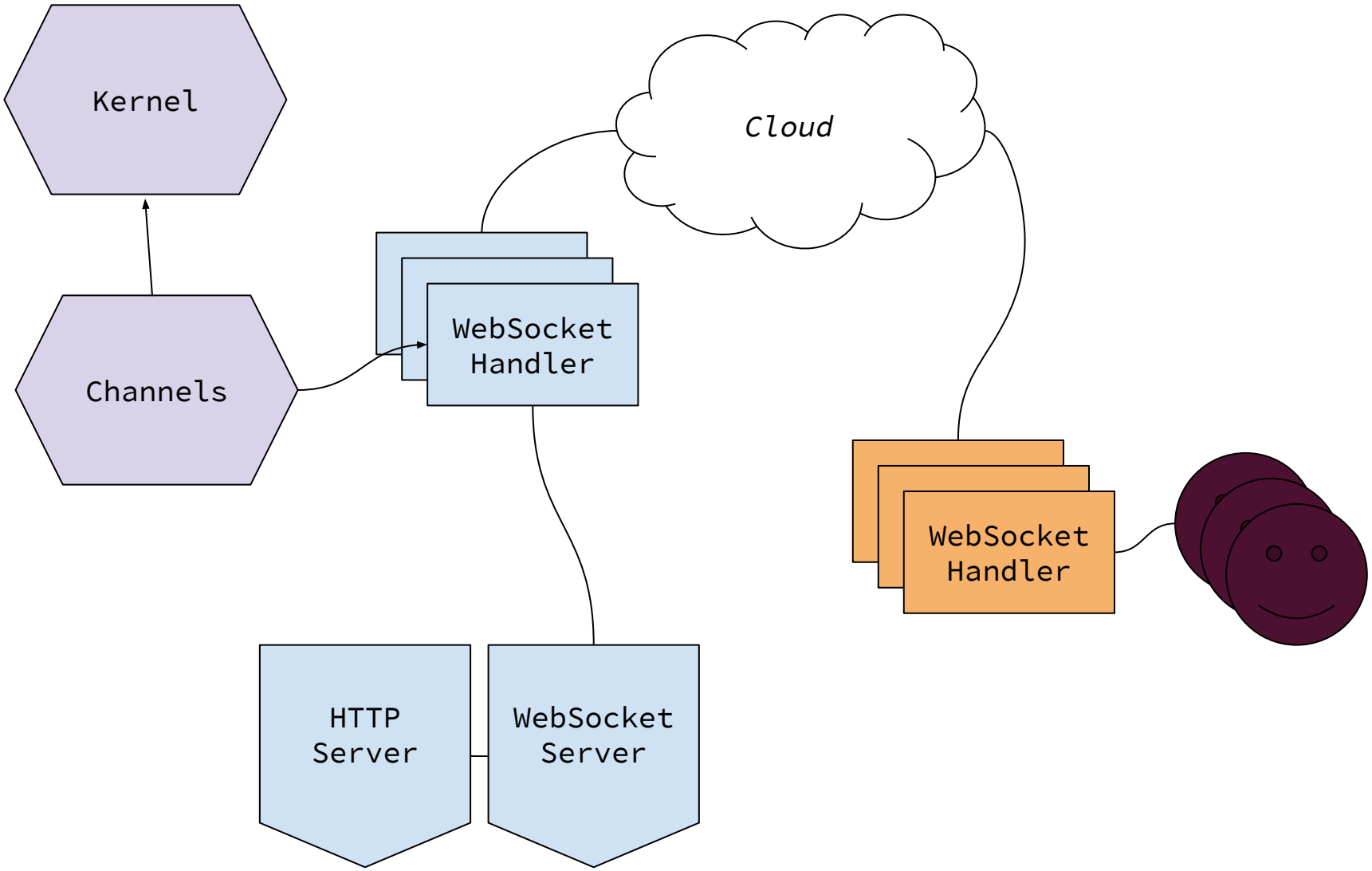
Buffered channels allow for multiple messages to be sent or received before blocking.

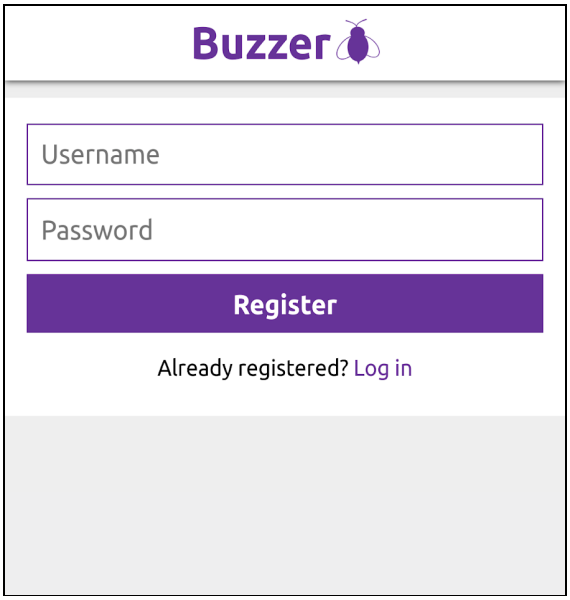
`select` statement operates on multiple channels. `default` case makes communication non-blocking.

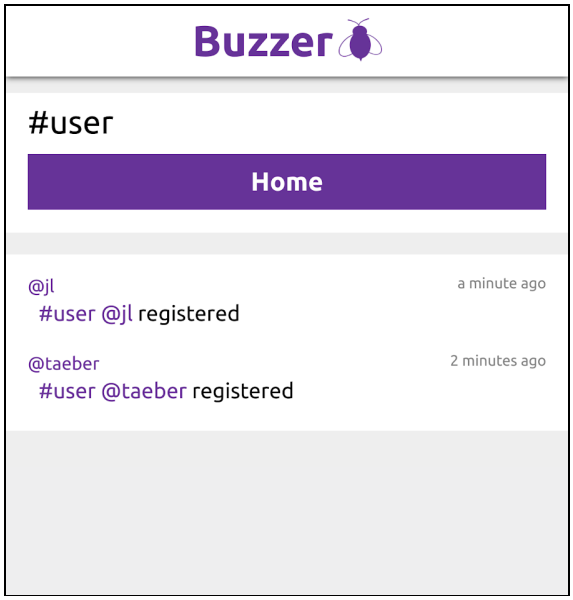
WebSockets

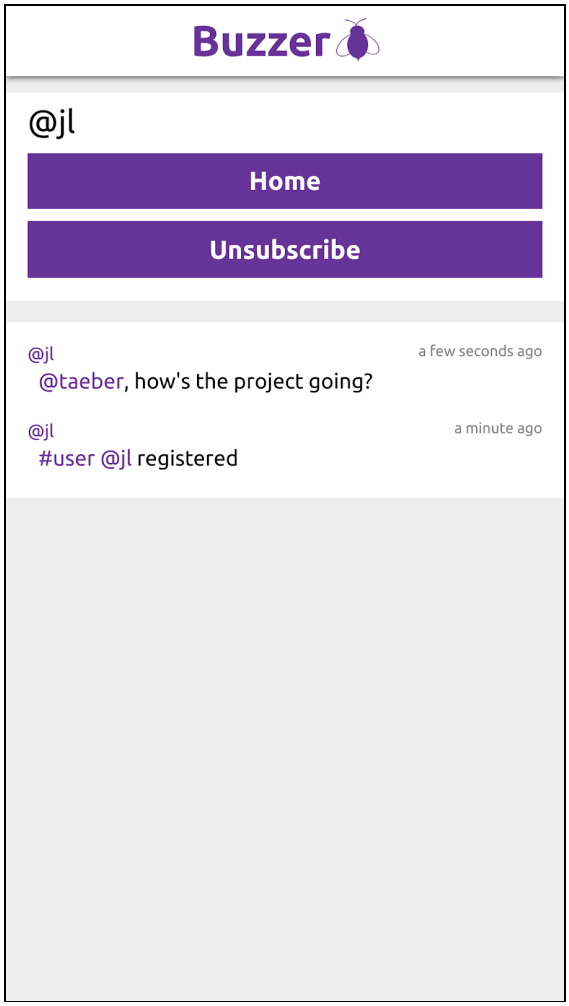
TCP-based protocol for bidirectional communication over ports 80 and 443 (for Transport Layer Security).

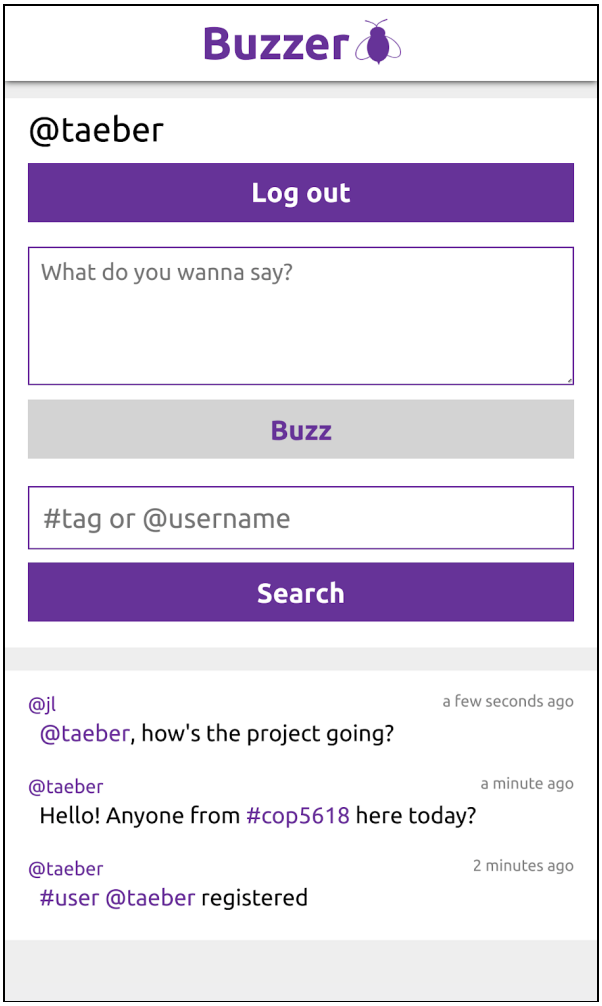
#Design


Kernel
Cloud
Channels
WebSocket
Handler
WebSocket
Handler
HTTP
Server
WebSocket
Server


Buzzer
Username
Password
Register
Already registered? Log in


Buzzer
#user
Home
@jl
a minute ago
#user @jl registered
@taeber
2 minutes ago
#user @taeber registered


Buzzer
@jl
Home
Unsubscribe
@jl
a few seconds ago
@taeber, how's the project going?
@jl
a minute ago
#user @jl registered


Buzzer
@taeber
Log out
What do you wanna say?
Buzz
#tag or @username
Search
@jl
a few seconds ago
@taeber, how's the project going?
@taeber
a minute ago
Hello! Anyone from #cop5618 here today?
@taeber
2 minutes ago
#user @taeber registered

#Locking

```
func newChannelServer(actual *kernel) *channelServer {
    // Use channels to sync access to actual kernel.
    return &channelServer{
        actual:   actual,
        post:     make(chan request, 100),
        follow:   make(chan request, 100),
        unfollow: make(chan request, 100),
        messages: make(chan request, 100),
...
        login:    make(chan request, 100),
        logout:   make(chan request, 100),
        shutdown: make(chan bool),
    }
}

func (server *channelServer) process() {
    for {
        select {
        case req := <-server.post:
            msgID, err := server.actual.Post(req)
            go respond(&req, response{data: msgID, error: err})
...
        case <-server.shutdown:
            return
        }
    }
}
```

#ConditionalSync

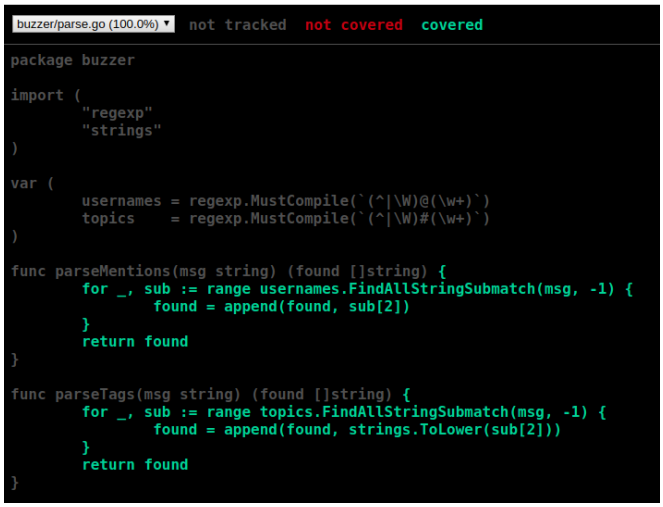
```
go func() { // Inside accept()
  defer func() { shutdown <- true }()
  for { // Processes any messages received.
    select { // Guarded suspension.
    case msg := <-received:
      client.decodeAndExecute(msg)
    case sub := <-client.subscribe:
      username := client.getUsername()
      if sub.follower != username {
        continue
      }
      if sub.unfollow {
        client.Write("unfollow "+sub.followee)
      } else {
        client.Write("follow " + sub.followee)
      }
    }
  }
}()
<-shutdown // Wait for shutdown.
```

#Semaphore

```
client := wsClient{
    username:  make(chan string, 1),
…
}

// client.username is used as a semaphore.
go func() {
    client.username <- ""
}()

func (client *wsClient) getUsername()
(username string) {
    username = <-client.username
    client.username <- username
    return
}

func (client *wsClient) setUsername(username
string) {
    <-client.username
    client.username <- username
}
```

#Tools

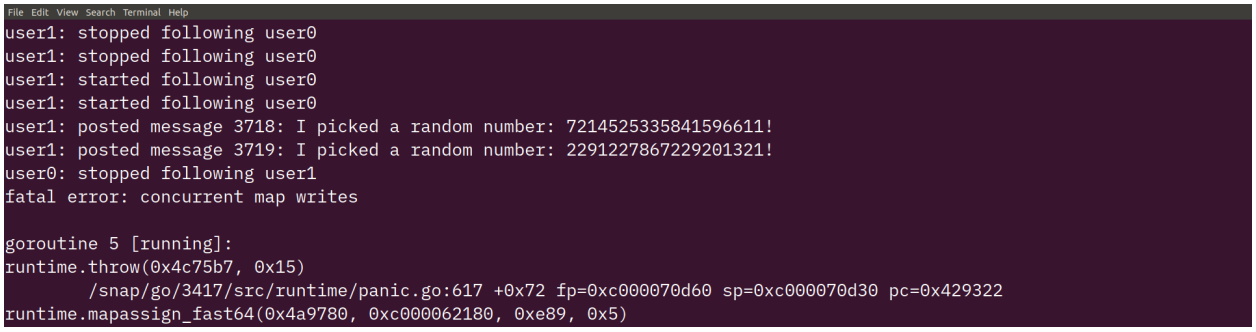
Test Runner, Code Coverage, and Benchmarking



Code Coverage output

```
$ make benchmark
GOPATH=/home/taeber/code/cop5618-concurrent/project/lib:/home/taeber/code/cop5618-concurrent/project go test -benchmem -run=^$ buzzer -bench .
...
BenchmarkKernelPost-4            300000   4415 ns/op   457 B/op       0 allocs/op
BenchmarkChannelServerPost-4     200000   6450 ns/op   534 B/op       3 allocs/op
PASS
ok     buzzer2.744s
```

Deadlock and race detection



Data race detection in Go runtime

#Conclusions

Goroutines are the agents.

Goroutines, Channels, and `select` provide an alternative to traditional shared memory concurrency concepts.

It is still easy to have concurrency issues/bugs when using channels.

The Go runtime and tools provides useful detection of data races and issues arising from non-determinate behavior.

Go's build tools provide convenient tooling to development software.

Go language constructs and library calls are used by the runtime to suspend Goroutines as needed.

#FutureWork

Implement the backend using more traditional shared memory primitives like locks and conditional variables to compare performance.

Write an Actor Model framework for Go or a language based on the Go library and runtime.

#Thanks

A Tour of Go, Golang.org Authors.
https://tour.golang.org/concurrency
WebSocket Protocol, Fette and Melnikov.
https://tools.ietf.org/html/rfc6455
Preparing and Presenting Effective Research Posters, Miller, Jane.
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1955747/

UF | UNIVERSITY of FLORIDA