The 1990s Called. They Want Their Code Back.

5 Ways Your Code is Stuck in the 90s

3 Mar 2015

Jonathan Oliver

$ whoami

@jonathan_oliver
http://jonathanoliver.com
http://github.com/joliver
http://keybase.com/joliver
Distributed Podcast
Chief SmartyPants, SmartyStreets

Overview

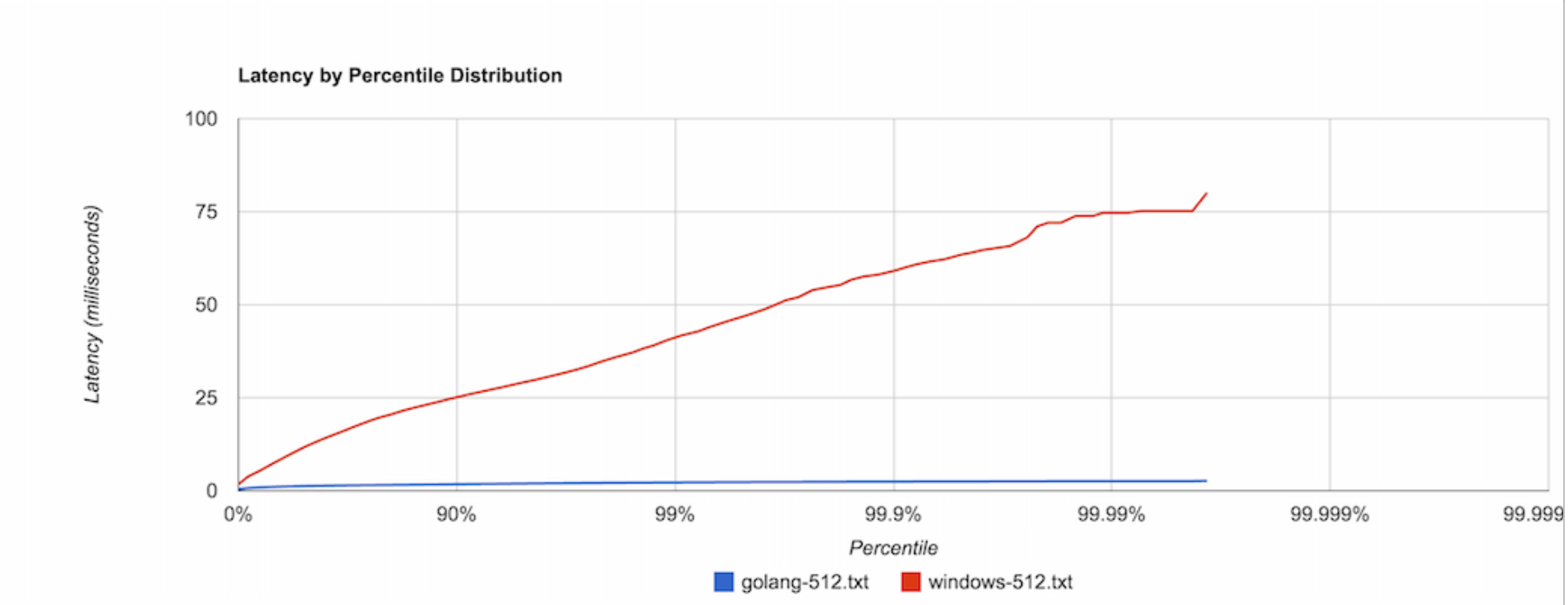
- Disclaimers
- Why Go Exists
- #1 Implicit messaging
- #2 Complex Threading
- #3 RPC Everywhere
- #4 GC
- #5 Logging
- Conclusion

Caveats: But, but, but...

- Your mileage may vary
- Don't apply blindly or wholesale
- Sharp knife

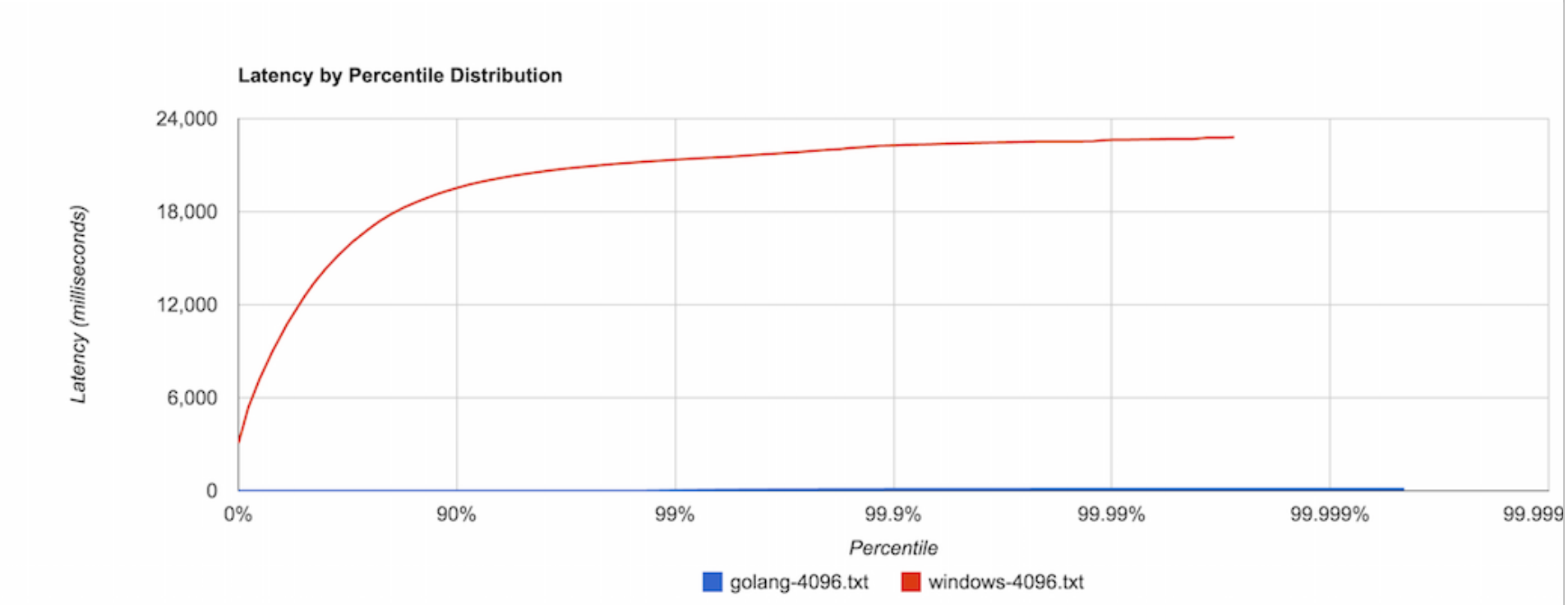
Our experience

- C# on Windows vs Go on Linux (512 req/s)



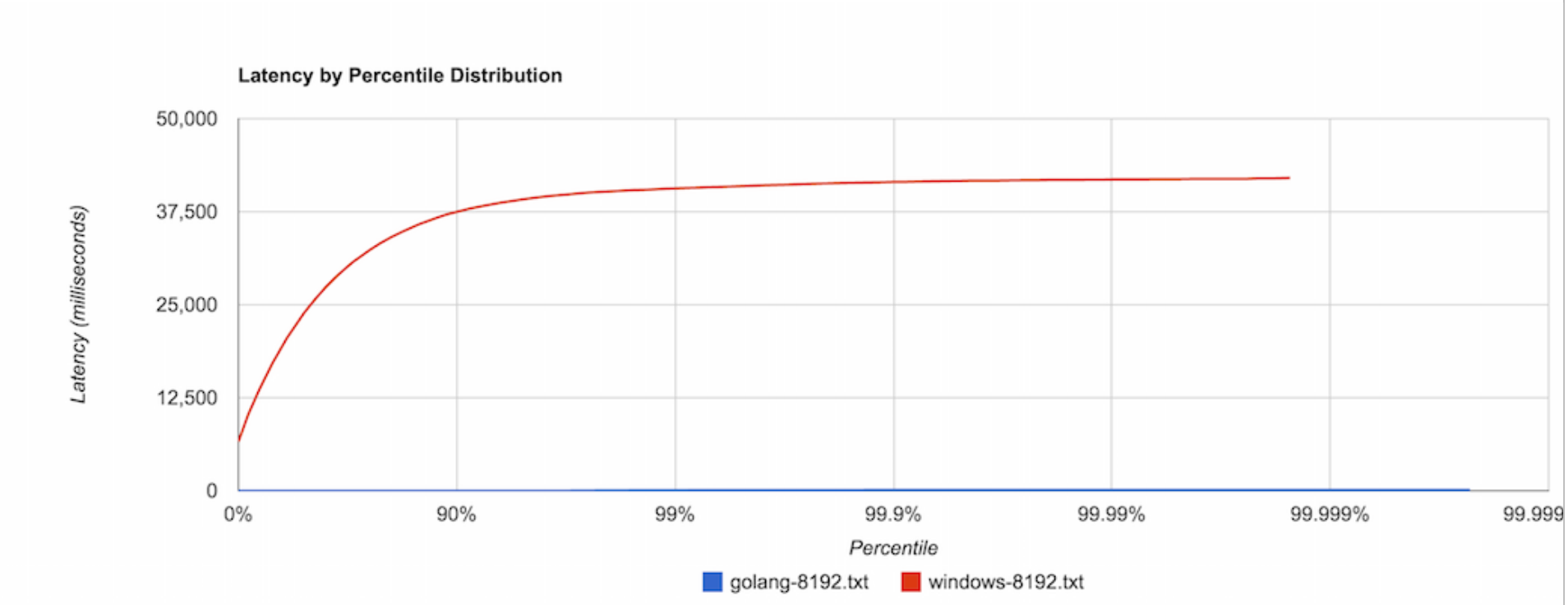
hdrhistogram.org

C# Windows vs Golang Linux at 4096 req/s



hdrhistogram.org

C# Windows vs Golang Linux at 8192 req/s



hdrhistogram.org

Why Go Exists

(The real reason)

- Compile times?
- Multi-core, networked systems

1990ism #1: Implicit Messaging

Hint Dropping Fallacy

- If you have to ask, it doesn't mean as much
- If you really loved me, you'd know

Implicit Messaging: PHP

```
<?php
    $sql = 'UPDATE Users SET ' +
       'firstname = ' $_GET['firstname'] + ','+
       'lastname = ' $_GET['lastname'] + ','+
       'phone = ' $_GET['phone'] + ','+
       'password = ' hash($_GET['password']) + ','+
       'WHERE id=' + $_GET['id'];
    mysql_query($sql, $connection) or die("Couldn't execute query.");
?>
```

Implicit Messaging: Go

- Where does HTTP stop and the application start?

```
func implicit(response http.ResponseWriter, request *http.Request) {
    query := request.URL.Query()

    statement := `UPDATE Users
        SET firstname = '%s',
        lastname = '%s',
        phone = '%s',
        password='%s'
        WHERE id = %s;`

    sql.Execute(statement,
        query.Get("firstname"),
        query.Get("lastname"),
        query.Get("phone"),
        hashAndSalt(query.Get("password")),
        query.Get("id"))

    response.WriteHeader(200)
}
```

Implicit Messaging: Boundaries

- HTTP bleeds all over the application
- .NET: System.Web.HttpContext.Current.Request...

Implicit Messaging: Intention?

- I know! I'll use a DTO that corresponds to my table!
- Hello, Ruby on Rails / Active Record

```
type User struct {
    ID        int
    FirstName string
    LastName  string
    Phone     string
    Password  []byte
}
```

- Staring at the table salt: implicit or inferred understanding


```
type User struct {
    ID        int
    FirstName string
    LastName  string
    Phone     string
    Password  []byte
}
```

Solution #1: Explicit Contracts

Application Protocols 101:

- HTTP: Hypertext **Transfer** Protocol
- SMTP: Simple Mail **Transfer** Protocol
- FTP: File **Transfer** Protocol (control channel, port 21)
- Transfering what?

- **Messages**!
- Review HTTP, SMTP, etc. RFC specifications
- e.g. HTTP **message** body, HTTP **message** headers, etc.
- HTTP, SMTP, etc. **encapsulate** a message

DTOs: What Are Your Intentions?

- Implicit / Inferred (Active Record)

```
type User struct {
    ID        int
    FirstName string
    LastName  string
    Phone     string
    Password  []byte
}
```

- Explicit

```
type ChangePasswordCommand struct {
    UserID               int
    NewPassword          string
    NewPasswordConfirmed string
    OldPassword          string
}
```

Messaging How-To

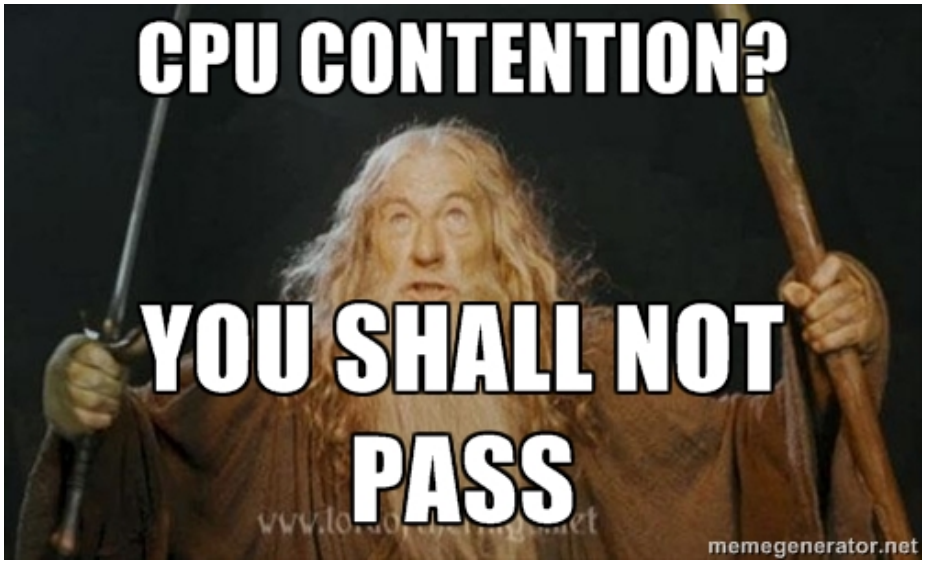
- HTTP values into message struct
- URL+VERB determines message type
- Query String
- Form Values
- Deserialize body or HTTP 400

Messaging How-To (continued)

- HTTP is an interface to application
- Push message into application layer
- Additional interfaces, e.g. SMTP, AMQP, CLI, etc.

1990ism #2: Complex Threading Code

- Goroutine per HTTP request
- Terrible for shared state like:
- Incrementing a counter
- Modify a map
- Updating object references



- Goroutine per request = manual synchronization of shared state
- Go doesn't save us from synchronization code
- go keyword can make things **harder**

```
package main

import "fmt"
import "time"

func main() {
    for i := 0; i < 4; i++ {
        go func() {
            fmt.Println(i) // bad closure
        }()
    }
    time.Sleep(time.Millisecond)
}
```

Solution #2: In-process "microservices" (Actors)

Actor Example:

```
// uncontended state

func listen() {
    for message := this.incomingChannel {

        // single-threaded with synchronization primitives
        counter++
        map[message.UserID]++

        // additional message processing code

        this.outgoingChannel <- message
    }
}
```

- The Unix Way: Small & **Composable**
- Message In, Message Out: Easy Testing
- Pipes and Filters
- Marshal to external process

Break Apart Stateful and Stateless Operations

```
func (this CounterPhase) listen() {
    for message := this.incomingChannel {
        counter++ // stateful; single-threaded with no sync code
        message.Sequence = counter
        this.outgoingChannel <- message // outgoing to process phase
    }
}
func (this ProcessPhase) listen() {
    // can be stateless because state was assigned in previous phase

    for i := 0; i < runtime.NumCPU(); i++ {
        go func() {
            for message := this.incomingChannel { // incoming from counter phase
                // process message (CPU/network operations)
                this.outgoingChannel <- message
            }
        }()
    }
}
```

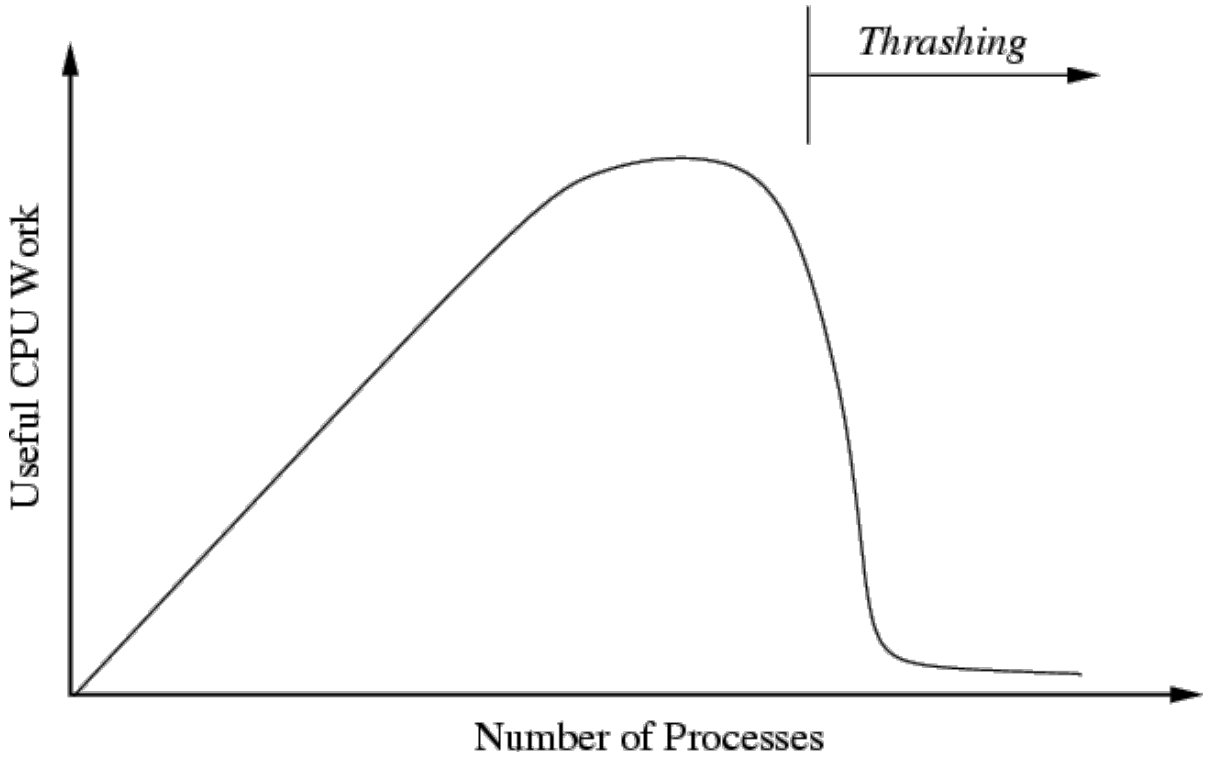
HTTP RPC

- Block the caller until the work is done

```
func handle(w http.ResponseWriter, r *http.Request) {
    var wg sync.WaitGroup
    wg.Add(1)

    query := r.URL.Query()
    this.application <- ChangePasswordCommand{
        UserID:               cookie.Get("user-id"),
        OldPassword:          query.Get("old-password"),
        NewPassword:          query.Get("new-password"),
        NewPasswordConfirmed: query.Get("new-password-confirmed"),
        WaitGroup:            &wg,
    }

    wg.Wait()

    // return result of application
}
```

Queues and Natural Backpressure

- Typical performance characteristics at 90% vs 99% utilization



1990ism #3: Remote Procedure Call Everywhere

Fallacies of Distributed Computing

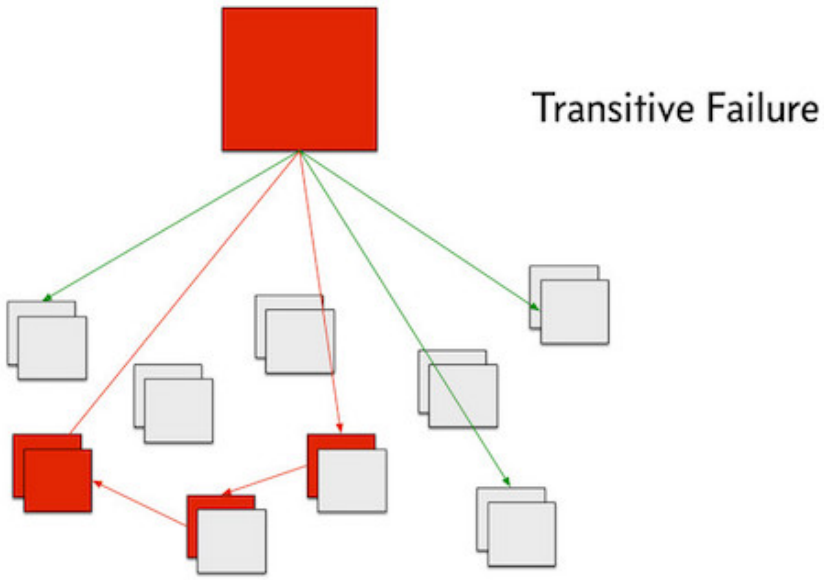
- The Network is Reliable
- Latency is Zero

Typical Application Behavior (Transaction Script)

- Opens a DB connection
- Start a transaction
- Execute DB operation(s)
- Other operations? (Send email, etc.)
- Commit transaction
- Wash, rinse, repeat
- What could possibly go wrong?

Fragile RPC

- Per business demands, we add "one more thing", e.g. email, etc.
- When network is down, lots of things break
- Bill credit card, send email, etc.
- Netflix architecture



Solution #3: Actors (again) + Embrace Failure

Simple Retry Code

```
import "time"

func listen() {
    // simple retry
    for message := range this.incoming {
        for attempt := 0; attempt < 5; attempt++ {
            if err := emailReceipt(message); err != nil {
                time.Sleep(time.Second * 30)
                continue
            }
        }
    }
}
```

BONUS POINTS: Simple Batching

- Story: Moving one box at a time

```
func listen() {
    for message := range this.incoming {
        addToUnitOfWork(message)
        if len(this.incoming) == 0 || len(batch) >= 100 {
            commit()
            newTransaction()
        }
    }
}
```

- 1-2 order of magnitude performance increase

1990ism #4: Abuse Garbage Collection

- Primitive, mark-and-sweep implementation
- But getting better...
- Java and .NET
- pointers
- maps
- strings
- slices

GC Pause Latency and You

- Are 500 ms GC pauses okay?
- How about 5 seconds?
- What is latency costing you?

Solution #4: Understanding GC Behavior

- Measure, measure, measure
- Avoid pointers (where possible)
- Preallocate and re-use structures (where possible)
- My bug report (issue #9477) & maps of structs (v1.5)
- Keep byte slices off heap (where possible)
- Size of the heap

1990ism #5: Logging Is Sufficient

- Logging is awesome, but very "trees" focused
- Stored?
- Where?
- How long?
- Who analyzes and when?
- Calls to `log.Print` result in blocking `syscalls` that yield the goroutine
- Hard to make blocking/yielding calls

Solution #5: Metrics, Metrics, Everywhere

Business Value (Coda Hale: Metrics, Metrics, Everywhere)

Business value is anything which makes people more likely to give us money

We want to generate more business value

Our code generates business value when it runs—NOT when we write it.

We need to make better decisions about our code

We need to know what our code does when it runs

We can’t do this unless we MEASURE it

Our mental model of our code is not our code.

Example: This code can’t possibly work; it works.

Example: This code can’t fail; it fails

Example: Do these changes make things faster?

We can’t know until we MEASURE it

We improve our mental model by measuring what our code DOES

A better mental model makes us better at deciding what to do—at generating business value


HTTP 200
1.20
1.00
0.80
0.60
0.40
0.20
0
Sat 4PM
Sat 8PM
API msecs (articles)
6.0 K
5.0 K
4.0 K
3.0 K
2.0 K
1.0 K
0
Sat 4PM
Sat 8PM
API msecs (subscriptions)
3500.0
3000.0
2500.0
2000.0
1500.0
1000.0
500.0
0
Sat 4PM
Sat 8PM
API msecs (folders)
1000.0
800.0
600.0
400.0
200.0
0
Sat 4PM
Sat 8PM
Memory used
8280.0 M
8160.0 M
8040.0 M
7920.0 M
7800.0 M
7680.0 M
Sat 4PM
Sat 8PM
RSS feeds queued
250.0
200.0
150.0
100.0
50.0
0
Sat 4PM
Sat 8PM
New RSS feeds
1.00
0.75
0.50
0.25
0
Sat 4PM
Sat 8PM
New articles
14.0
12.0
10.0
8.0
6.0
4.0
2.0
Sat 4PM
Sat 8PM
RSS feed errors
0.70
0.60
0.50
0.40
0.30
0.20
0.10
Sat 4PM
Sat 8PM
Network recv (bytes)
2250.0 K
2000.0 K
1750.0 K
1500.0 K
1250.0 K
1000.0 K
750.0 K
500.0 K
250.0 K
0
Sat 4PM
Sat 8PM
RSS feeds retrieved
15.0
13.5
12.0
10.5
9.0
7.5
6.0
4.5
Sat 4PM
Sat 8PM
Total RSS feeds
102389.00
102388.75
102388.50
102388.25
102388.00
Total articles
15500.0 K
15475.0 K
15450.0 K
15425.0 K
15400.0 K
15375.0 K
15350.0 K
15325.0 K
RSS feeds skipped
2.00
1.75
1.50
1.25
1.00
0.75
0.50
Sat 4PM
Sat 8PM
Network sent (bytes)
1000.0 K
875.0 K
750.0 K
625.0 K
500.0 K
375.0 K
250.0 K
125.0 K
Sat 4PM
Sat 8PM
IO read (bytes)
840.0 M
720.0 M
IO read (time)
3000.0
2500.0
IO write (bytes)
3000.0 K
2500.0 K
IO write (time)
480.0
360.0
Disk space %
100.0
75.0

Understanding Your Application

- Instrument your application (metrics)
- Understand how it's being used
- Understand the pathways that are taken (counters)
- Understand how much (disk/memory/etc) you have left (gauges)

Service Providers

- Librato (http://github.com/smartystreets/metrics)
- Boundary
- Datadog

Key Takeaways

- Go != other languages
- Work with concurrency primitives
- Explicit messages
- Message pipelines ("actors")
- Simple logic, simple code

Thank you

3 Mar 2015

Jonathan Oliver
@jonathan_oliver (http://twitter.com/jonathan_oliver)

http://jonathanoliver.com (http://jonathanoliver.com)
http://github.com/joliver (http://github.com/joliver)
http://keybase.com/joliver (http://keybase.com/joliver)
Distributed Podcast
Chief SmartyPants, SmartyStreets