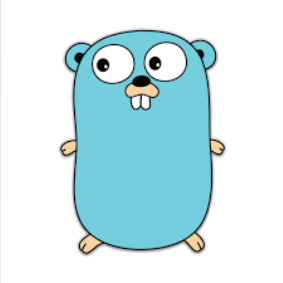
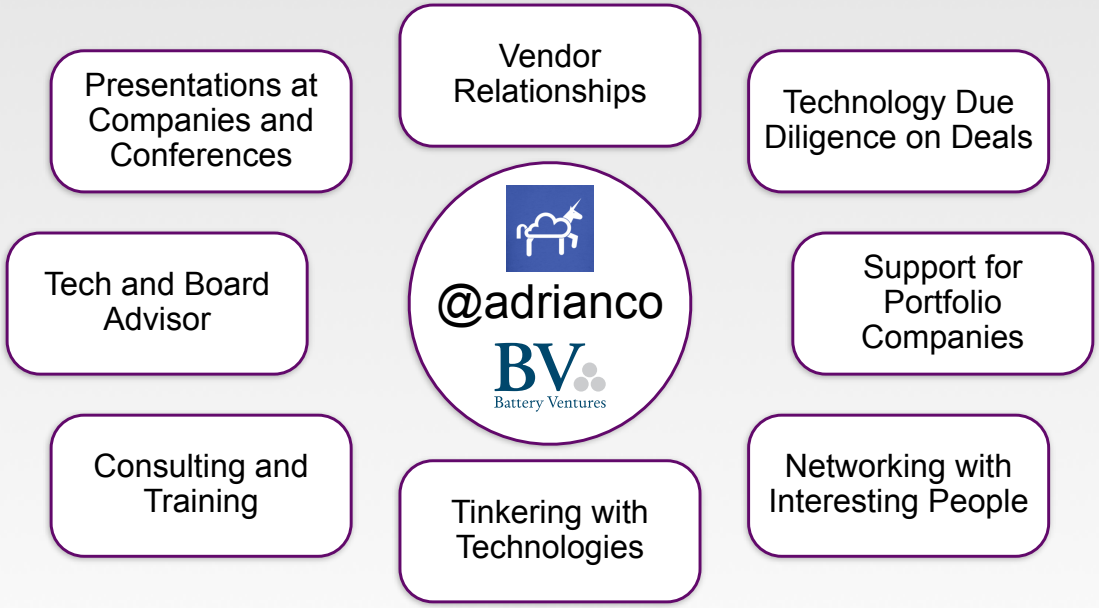
Communicating Sequential Goroutines

Adrian Cockcroft @adrianco
Technology Fellow - Battery Ventures
July 2016

What does @adrianco do?





Previously: Netflix, eBay, Sun Microsystems, Cambridge Consultants, City University London - BSc Applied Physics

Agenda

1978 *Communicating Sequential Processes*
1983 *Occam*
How Channels Work
1992 *Pi-Calculus*
2013 *The Life of Occam-Pi*
2006 *Occam-Pi based simulation*
Pi-Calculus ideas in Go
Go Applications

What is CSP?

Programming Techniques
S. L. Graham, R. L. Rivest Editors

Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.

Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays
CR Categories: 4.20, 4.22, 4.32

"...the concepts and notations introduced in this paper (although described in the next section in the form of a programming language fragment) ***should not be regarded as suitable for use as a programming language,*** *either for abstract or for concrete programming. They are at best only a partial solution to the problems tackled."*

Subset of CSP Syntax

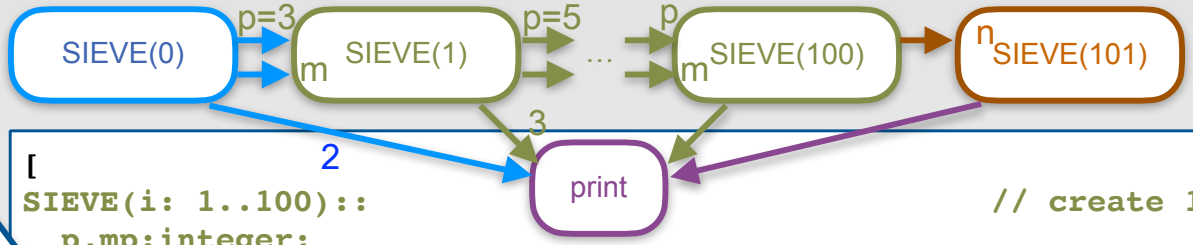
Note: Named channels are absent

```
<command> ::= <simple command>|<structured command>
<simple command> ::= <null command>|<assignment command>|<input command>|<output command>
<structured command> ::= <alternative command>|<repetitive command>|<parallel command>
<null command> ::= skip
<command list> ::= {<declaration>;|<command>;} <command>
<assignment command> ::= <target variable> := <expression>
<input command> ::= <source>?<target variable>
<output command> ::= <destination>!<expression>
<source> ::= <process name>
<repetitive command> ::= *<alternative command>
<alternative command> ::= [<guarded command> {[]<guarded command>}]
<guarded command> ::= <guard> → <command list>
<guard> ::= <guard list>|<guard list>;<input command>|<input command>
<parallel command> ::= [<process>{||<process>}]
```

named process addressing

There will be a test…

Prime Number Sieve in CSP



read ?p

send !p

Repetitive *

Alternate [ ]

Parallel | |

Sequential ;

```
[
SIEVE(i: 1..100)::                                  // create 100 sieve processes
  p,mp:integer;
  SIEVE(i - 1)?p;                                   // first thing read is prime
  print!p;                                          // send it to be printed
  mp := p;                                          // mp is a multiple of p
  *[m:integer; SIEVE(i - 1)?m →                     // for loop wait for next
    *[m > mp → mp := mp + p];                       // for m > mp { mp += p }
    [m = mp → skip [] m < mp → SIEVE(i + 1)!m]      // if nonprime skip else send next
  ]
||SIEVE(0)::print!2; n:integer; n := 3;             // start here with 2
  *[n < 10000 → SIEVE(1)!n; n := n + 2]             // for n=3;n<10000;n+=2 {send n}
||SIEVE(101)::*[n:integer;SIEVE(100)?n → print!n]   // last prime in line
||print::*[(i:0..101) n:integer; SIEVE(i)?n → …]    // print all the things somehow…
]
```

I warned you there was a test!

CSP Issues:

Not a full language
Hard to read
Process addressing

Occam's Razor

Non sunt multiplicanda entia sine necessitate

"Entities must not be multiplied beyond necessity"

William of Ockham (c. 1287–1347)

David May's Occam Language

Extremely simple and elegant
implementation of CSP as a language

Adds named channels
Designed as the assembly language for Transputer hardware

```
Occam is intended to be the smallest language which  is  adequate
for  its purpose; however, suggestions for further simplification
would be welcome.
```

by David May

INMOS Limited
Whitefriars
Lewins Mead
Bristol BS1 2NP
UK
(0272) 290861
Telex 444723

INMOS Corporation
P O Box 16000
Colorado Springs
Colorado 80935
USA
(303) 630-4000
TWX 910 920 4904

1. Introduction

VLSI technology allows a whole microcomputer with memory, processor and communications to be constructed as a single device. Such a device can be used as a component in the construction of high performance systems. It provides a 'building block' for such systems, in the same way that logic gates provide a 'building block' for digital systems.

New programming languages are needed to allow systems consisting of many interconnected microcomputers to be designed and programmed. Occam is such a language, developed by INMOS. It is a simple language based on the concepts of concurrency and communication. It may be used both to describe the structure of a system in terms of connected microcomputers, and to program the individual microcomputers.

An initial version of occam has been produced, and this is described below. Further information, and an implementation of occam, is available from INMOS.

SIGPLAN Notices, Vi8 #4, April, 1983

Subset of Occam Syntax

Indent based structure

Definitions scoped to block following `:`

Reserved words in upper case

Time is a built in concept

Only has VAR and CHAN types

```
PROC delay(CHAN stop, in, out)
  VAR going, v, d:
  SEQ
    going := TRUE
    WHILE going
      ALT
        in?v
          d := NOW + 100
        WAIT AFTER d
          out!v
        stop?ANY
          going := FALSE
:
```

ALTernate blocks on multiple input or time guards

Only has PROC hence non-functional programming!

```
PROC pipeline(CHAN in, out)
  CHAN c[101]:
  PAR
    VAR x:
    SEQ
      in?x
      c[0]!x
    PAR i=[1 FOR 100]
      VAR y:
      SEQ
        c[i-1]?y
        c[i]!y+1
    VAR z:
    SEQ
      c[100]?z
      out!z
:
```

PARallel runs every line in parallel and waits

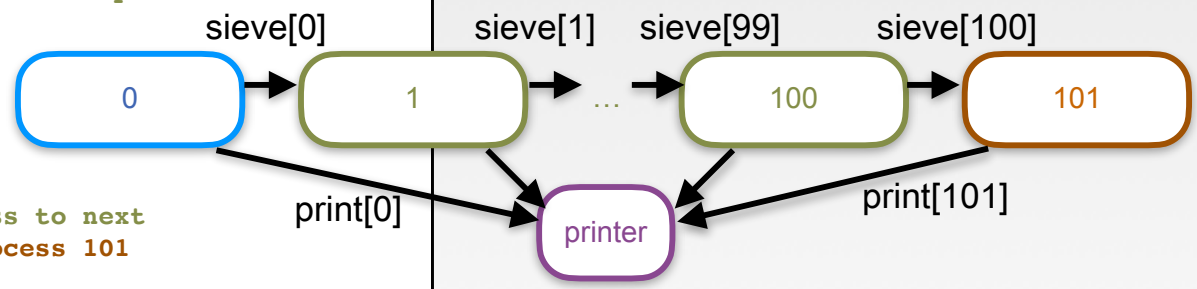
FOR replicator can be used with SEQ, PAR, ALT, IF

IF (not shown) like ALT with test guards

Disclaimer: This version of the Occam language was written in Pascal on VAX/VMS. These code examples have only been checked by manual inspection

```
PROC eratosthenes(CHAN output)
  CHAN sieve[101], print[102]:
  PAR
    SEQ                                 // process 0
      print[0]!2                        // start primes with 2
      SEQ n = [1 FOR 4999]              // for n=3;n<10000;n+=2 {send n}
        sieve[0]!(n+n)+1                // explicit evaluation order
      sieve[0]!-1                       // send -1 as terminator
    PAR i = [1 FOR 100]                 // create 100 sieve processes
      var p, mp, m:
      SEQ
        sieve[i-1]? p                   // first thing read is prime
        print[i]! p                     // send it to be printed
        mp := p                         // mp is a multiple of p
        WHILE m <> -1                   // look for terminator
          SEQ
            sieve[i-1]? m               // read from previous
            WHILE m > mp
              mp := mp + p
              IF
                m = mp
                  SKIP
                m < mp
                  sieve[i]! m           // pass to next
    var n:                              // process 101
    SEQ
      sieve[100]? n
      print[101]! n                     // last prime in line
      sieve[100]? ANY                   // discard terminator
    var n:                              // printer process
    SEQ i= [0 FOR 101]
      SEQ
        sieve[i]? n
        output! n
:
```

Prime Number Sieve Translated from CSP to Occam



Naming shifts from processes to channels

Channels behave like distributed assignment

How does a CHAN work?

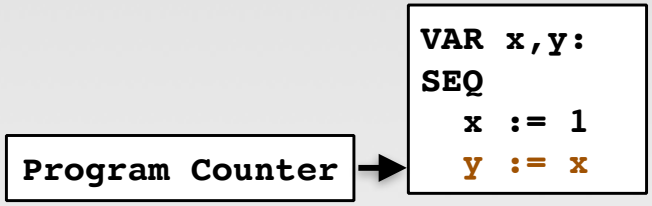
Comparing Occam and Go

Implementation of simple assignment is shown first as a baseline example

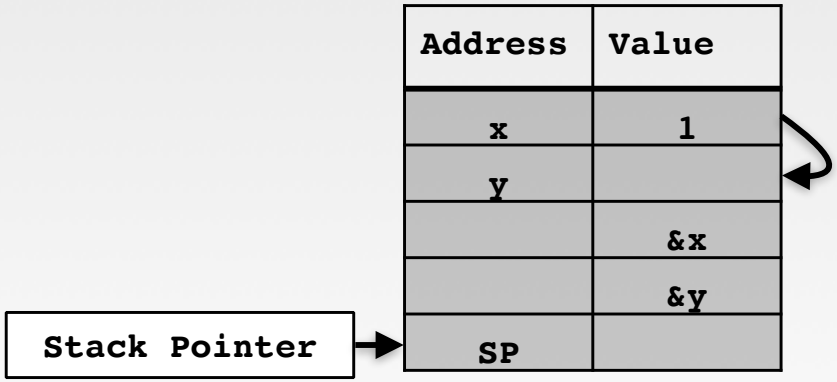
```
PROC main(chan out)
  VAR x,y:
  SEQ
    x := 1
    y := x
    out!y
:
```

```
func main() {
        var y int
        x := 1
        y = x
        fmt.Println(y)
}
```

How does assignment work?



Implementation of simple assignment



Occam assumes a simple stack machine

```
push &x
push &y
copy        // *(SP+2) to *(SP+1)
```

Stack Points to an empty location that can be used to save the PC if this proc pauses

Comparing Occam and Go

Parallel Channel Assignment

```
PROC main(CHAN out)
  VAR x,y:
  SEQ
    x := 1
    CHAN c:
    PAR
      c!x
      c?y
    out!y
:
```

```
func main() {
        var x, y int
        x = 1
        c := make(chan int)
        var wg sync.WaitGroup
        wg.Add(2)
        go func() { defer wg.Done(); c <- x }()
        go func() { defer wg.Done(); y = <-c }()
        wg.Wait()
        fmt.Println(y)
}
```

How does a CHAN work?

Implementation of parallel channel assignment

```
VAR x,y:
SEQ
  x := 1
  CHAN c:
  PAR
    c!x
    c?y
```

Program Counter X

Program Counter Y

Address	Value
x	1
y	
c	SPX
	&x
SPX	PCX
	&y
SPY	

Stack Pointer X

Stack Pointer Y

First proc pushes &x then test-and-sets the channel, which is empty, so it writes stack pointer into channel, writes program counter to stack and sleeps

```
push &x
c==nil so c=SPX, *SPX=PCX, sleep
```

Second proc pushes &y then test-and-sets channel, which is full, so read other stack pointer from channel, grab x from it and store to y. Put other proc stack on run queue and clear channel for next time.

```
push &y
c!=nil so read *c+1 // into &y
run *c, c=nil
```

Disclaimer: This is intended to be a useful mental model for how channels work, not an exact implementation. Transputers did this in hardware!

Pi-Calculus
Robin Milner 1992

A Calculus of Mobile Processes, I

ROBIN MILNER

University of Edinburgh, Scotland

JOACHIM PARROW

Swedish Institute of Computer Science, Kista, Sweden

AND

DAVID WALKER

University of Warwick, England

We present the π-calculus, a calculus of communicating systems in which one can naturally express processes which have changing structure. Not only may the component agents of a system be arbitrarily linked, but a communication between neighbours may carry information which changes that linkage. The calculus is an extension of the process algebra CCS, following work by Engberg and Nielsen, who added mobility to CCS while preserving its algebraic properties. The π-calculus gains simplicity by removing all distinction between variables and constants; communication links are identified by *names*, and computation is represented purely as the communication of names across links. After an illustrated description of how the π-calculus generalises conventional process algebras in treating mobility, several examples exploiting mobility are given in some detail. The important examples are the encoding into the π-calculus of higher-order functions (the λ-calculus and combinatory algebra), the transmission of processes as values, and the representation of data structures as processes. The paper continues by presenting the algebraic theory of *strong bisimilarity* and *strong equivalence*, including a new notion of equivalence indexed by *distinctions*—i.e., assumptions of inequality among names. These theories are based upon a semantics in terms of a labeled transition system and a notion of *strong bisimulation*, both of which are expounded in detail in a companion paper. We also report briefly on work-in-progress based upon the corresponding notion of *weak* bisimulation, in which internal actions cannot be observed.

We present the π-calculus, a calculus of communicating systems in which one can naturally express processes which have changing structure. Not only may the component agents of a system be arbitrarily linked, but a communication between neighbours may carry information which changes that linkage.

In this paper we do not present the basic semantics of the calculus; this is done in our companion paper (Milner, Parrow, and Walker, 1989), in the same style as in CCS, namely as a labeled transition system defined by structural inference rules. In that paper the notions of strong bisimulation and strong equivalence are also defined; the latter is a congruence relation, so it may be understood as (strong) semantic equality. Here, we shall rely somewhat upon analogy with the transitions of CCS agents. In particular, we assume simple transitions such as

$$(\cdots + \bar{y}x.P + \cdots) \mid (\cdots + y(z).Q + \cdots) \xrightarrow{\tau} P \mid Q\{x/z\}$$

and simple equations such as

$$(y)(\bar{y}x.P \mid y(z).Q) = \tau.(y)(P \mid Q\{x/z\})$$

It's easy to show that...

This paper is incomprehensible!

A triumph of notation over comprehension.

Simple equations such as...

It is illuminating to see how the encoding of a particular example behaves. Consider $(\lambda x x)N$; first, we have

$$[\![\lambda x x]\!]\, v \equiv v(x)(w).\bar{x}w.$$

So, assuming x not free in N,

$$\begin{aligned}
[\![(\lambda x x)N]\!]\, u &\equiv (v)([\![\lambda x x]\!]\, v \mid (x)\bar{v}xu.x(w).[\![N]\!]\, w) \\
&\equiv (v)(v(x)(w).\bar{x}w \mid (x)\bar{v}xu.x(w).[\![N]\!]\, w) \\
&= \tau.(v)(x)(v(w).\bar{x}w \mid \bar{v}u.x(w).[\![N]\!]\, w) \\
&= \tau.\tau.(v)(x)(\bar{x}u \mid x(w).[\![N]\!]\, w) \\
&= \tau.\tau.\tau.(v)(x)(\mathbf{0} \mid [\![N]\!]\, u) = \tau.\tau.\tau.[\![N]\!]\, u.
\end{aligned}$$

More generally, it is easy to show that

$$[\![(\lambda x M)N]\!]\, u \approx [\![M\{N/x\}]\!]\, u, \qquad (36)$$

The Life of Occam-Pi
Peter Welch 2013

(a wonderful and easy to read history of parallel languages)

Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Life of occam-Pi

Peter H. WELCH

School of Computing, University of Kent, UK

p.h.welch@kent.ac.uk

Abstract. This paper considers some questions prompted by a brief review of the history of computing. Why is programming so hard? Why is concurrency considered an *"advanced"* subject? What's the matter with *Objects*? Where did all the Maths go? In searching for answers, the paper looks at some concerns over fundamental ideas within *object orientation* (as represented by modern programming languages), before focussing on the concurrency model of communicating processes and its particular expression in the occam family of languages. In that focus, it looks at the history of occam, its underlying philosophy (*Ockham's Razor*), its semantic foundation on Hoare's CSP, its principles of *process oriented design* and its development over almost three decades into occam-π (which blends in the concurrency dynamics of Milner's π-calculus). Also presented will be an urgent need for rationalisation – occam-π is an experiment that has demonstrated significant results, but now needs time to be spent on careful review and implementing the conclusions of that review. Finally, the future is considered. In particular, is there a future?

Keywords. process, object, local reasoning, global reasoning, occam-pi, concurrency, compositionality, verification, multicore, efficiency, scalability, safety, simplicity

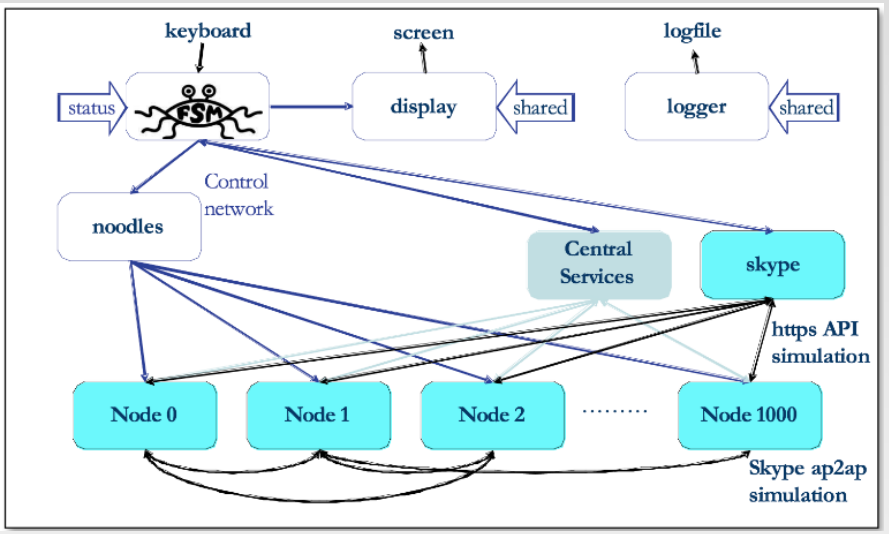
...looks at the history of occam, its underlying philosophy (Ockham's Razor), its semantic foundation on Hoare's CSP, its principles of process oriented design and its development over almost three decades into occam-π (which blends in the concurrency dynamics of Milner's π-calculus).

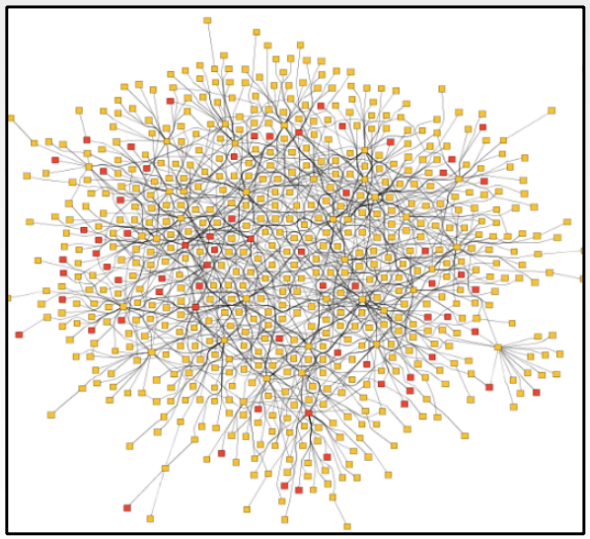
Simulation of Skype Peer-to-peer Web Services Choreography Using Occam-Pi

Adrian Cockcroft 2006

https://github.com/adrianco/spigo/blob/master/SkypeSim07.pdf

Occam-Pi was used to create an actor-like protocol simulator with supervisor pattern and dynamic channel based connections between nodes



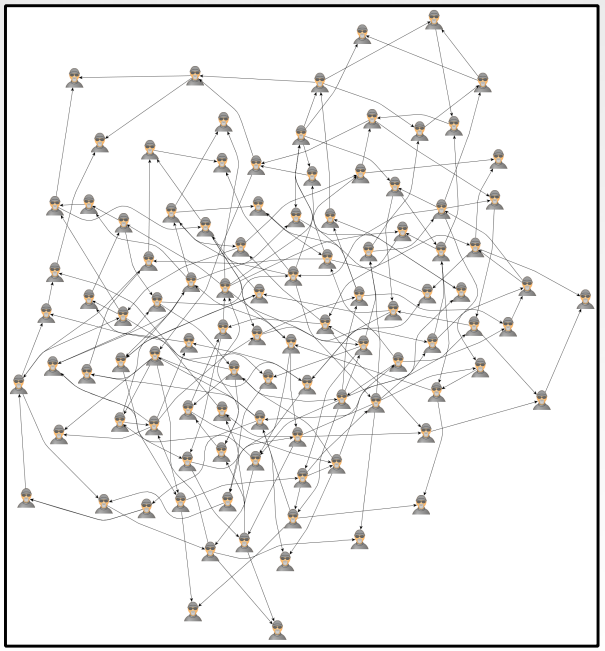


```
PROC lambda (CHAN BYTE kyb?, scr!)   -- connection to stdio of running process
  ...  declarations
  SEQ
    -- magically create many to one channels
    d.shared, d.chan := MOBILE MDISP
    api.put.shared, api.put.chan := MOBILE API.CALL.CHAN
    sc.shared, sc.chan := MOBILE SKYPE.CALL.CHAN
    chat.to.fsm.shared, chat.to.fsm.chan := MOBILE LISTEN.NODE.CHAN
    PAR
      display(d.chan?, scr!)
      fsm(kyb?, d.shared!, fsm.command!, chat.to.fsm.chan?, fsm.to.services!, fsm.to.skype!)
      noodles(fsm.command?, noodly.touch!)
      services(d.shared!, api.put.chan?, api.get!, fsm.to.services?)
      skype(d.shared!, sc.chan?, skype.response!, fsm.to.skype?)
      PAR i = 0 FOR num.nodes
        node (i, sc.shared!, skype.response[i]?, api.put.shared!, api.get[i]?,
          noodly.touch[i]?, chat.to.fsm.shared!)
:
```

Pi-Calculus Patterns in Go

Ported framework and FSM/Pirate P2P trading from Occam-Pi to Go

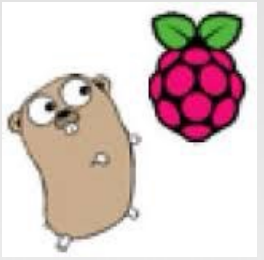
https://github.com/adrianco/spigo/tree/master/tooling/fsm
https://github.com/adrianco/spigo/tree/master/actors/pirate
https://github.com/adrianco/spigo/tree/master/benchmarks



```
$ ./spigo.1.6.2 -d=0 -p=100000 -a fsm -cpuprofile fsm.go162.profile
2016/04/20 23:55:50 fsm: population 100000 pirates
2016/04/20 23:55:51 fsm: Talk amongst yourselves for 0
2016/04/20 23:55:55 fsm: Delivered 500000 messages in 4.266286041s
2016/04/20 23:55:55 fsm: Shutdown
2016/04/20 23:55:56 fsm: Exit
2016/04/20 23:55:56 spigo: complete

$ ./spigo.1.6.2 -a fsm -d 0 -p 100000
2016/04/21 20:04:05 fsm: Delivered 500000 messages in 3.61466773s

$ ./spigo.1.5.1 -a fsm -d 0 -p 100000
2016/04/21 20:03:49 fsm: Delivered 500000 messages in 3.327230366s

$ ./spigo.1.4.2 -a fsm -d 0 -p 100000
2016/04/21 20:03:38 fsm: Delivered 500000 messages in 2.547419582s
```

Why is Go getting slower?

Go-π



Dynamic Channel Protocol
Actor Pattern
Partitioned Service Registry
Logging and Tracing

Dynamic Channel Protocol

https://github.com/adrianco/spigo/tree/master/tooling/gotocol

Imposition/Intention? https://en.wikipedia.org/wiki/Promise_theory

```
ch <-gotocol.Message{gotocol.GetRequest, listener, now, ctx, "why?"}

// Message structure used for all messages, includes a channel of itself
type Message struct {
        Imposition   Impositions  // request type
        ResponseChan chan Message // place to send response messages
        Sent         time.Time    // time at which message was sent
        Ctx          Context      // message context
        Intention    string       // payload
}
```

```
// Impositions is the promise theory term for requests made to a service
type Impositions int

// Constant definitions for message types to be imposed on the receiver
const (
            // Hello ChanToParent Name Initial noodly touch to set identity
            Hello Impositions = iota
            // NameDrop ChanToBuddy NameOfBuddy Here's someone to talk to
            NameDrop
            // Chat - ThisOften Chat to buddies time interval
            Chat
            // GoldCoin FromChan HowMuch
            GoldCoin
            // Inform loggerChan text message
            Inform
            // GetRequest FromChan key Simulate http inbound request
            GetRequest
            // GetResponse FromChan value Simulate http outbound response
            GetResponse
            // Put - "key value" Save the key and value
            Put
            // Replicate - "key value" Save a replicated copy
            Replicate
            // Forget - FromBuddy ToBuddy Forget link between two buddies
            Forget
            // Delete - key Remove key and value
            Delete
            // Goodbye - name - tell supervisor and exit
            Goodbye // test assumes this is the last and exits
            numOfImpositions
)
```

Dynamic Channel Protocol "Impositions"

Elements of DHCP + DNS + HTTP + Routing + Payment

Spigo Actor Pattern

```
func Start(listener chan gotocol.Message) {
        ...
        for {
                select {
                case msg := <-listener:
                        flow.Instrument(msg, name, hist)
                        switch msg.Imposition {
                        case gotocol.Hello:      // get named by parent
                                ...
                        case gotocol.NameDrop:   // someone new to talk to
                                ...
                        case gotocol.Put: // upstream request handler
                                ...
                                outmsg := gotocol.Message{gotocol.Replicate, listener, time.Now(),
                                 msg.Ctx.NewParent(), msg.Intention}
                                flow.AnnotateSend(outmsg, name)
                                outmsg.GoSend(replicas)
                        }
                case <-eurekaTicker.C:           // poll the service registry
                        ...
                }
        }
}
```

Instrument incoming requests

update trace context

Instrument outgoing requests

Skeleton code for replicating a Put message

Partitioned/Scoped Service Registries

```
// Package eureka is a service registry for the architecture configuration (nodes) as it evolves
// and passes data to edda for logging nodes and edges
package eureka

func Start(listener chan gotocol.Message, name string) {
     // lots of code removed
     for {
          msg = <-listener
          collect.Measure(hist, time.Since(msg.Sent))
          switch msg.Imposition {
          // lots more cases removed
          case gotocol.GetRequest:
               for n, ch := range microservices {
                    // respond with all the online names that match the service component
                    if names.Service(n) == msg.Intention {
                         // if there was an update for the looked up service since last check
                         if metadata[n].registered.After(lastrequest[callback{n, msg.ResponseChan}]) {
                              if metadata[n].online {
                                   gotocol.Message{gotocol.NameDrop, ch, time.Now(), gotocol.NilContext,
                                        n}.GoSend(msg.ResponseChan)
                              } else {
                                   gotocol.Message{gotocol.Forget, ch, time.Now(), gotocol.NilContext,
                                        n}.GoSend(msg.ResponseChan)
                              }
                         }
                         // remember for next time
                         lastrequest[callback{n, msg.ResponseChan}] = msg.Sent
                    }
               }
          }
     }
}
```

Registry is named at startup by supervisor

Service registry lookup handler GetRequest “genericname”

Only respond if there’s a change since this client last asked

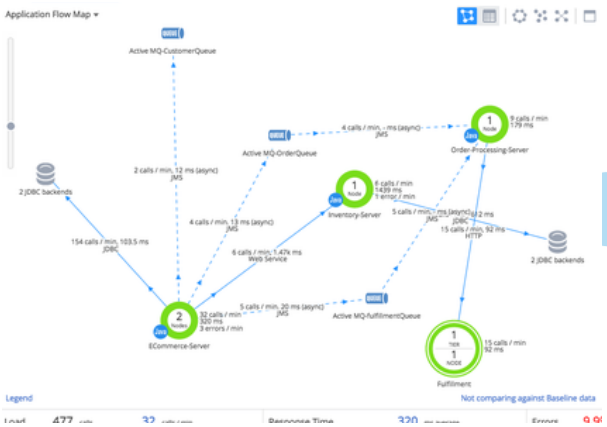
Service registry lookup returns NameDrop chan “instancename” or Forget chan “instancename”

What else could a golang actor-like simulator be useful for?

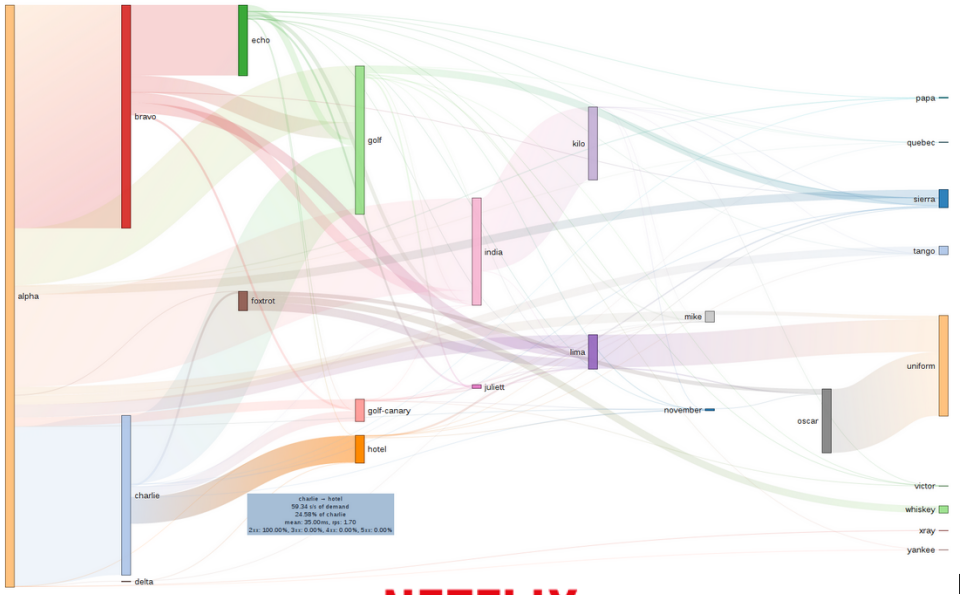
(slides from my Microservice talks)

Challenges for Microservice Platforms

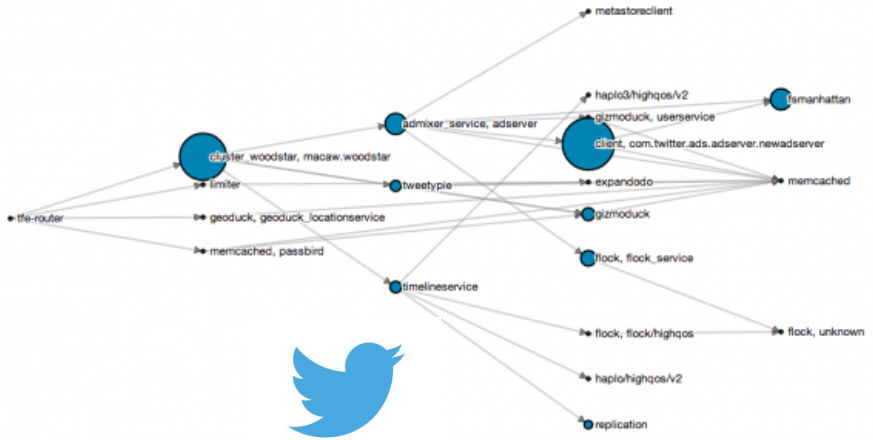
Some tools can show the request flow across a few services



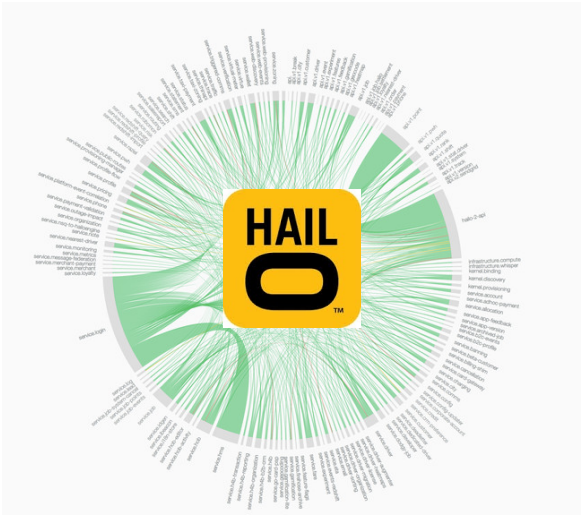
APPDYNAMICS

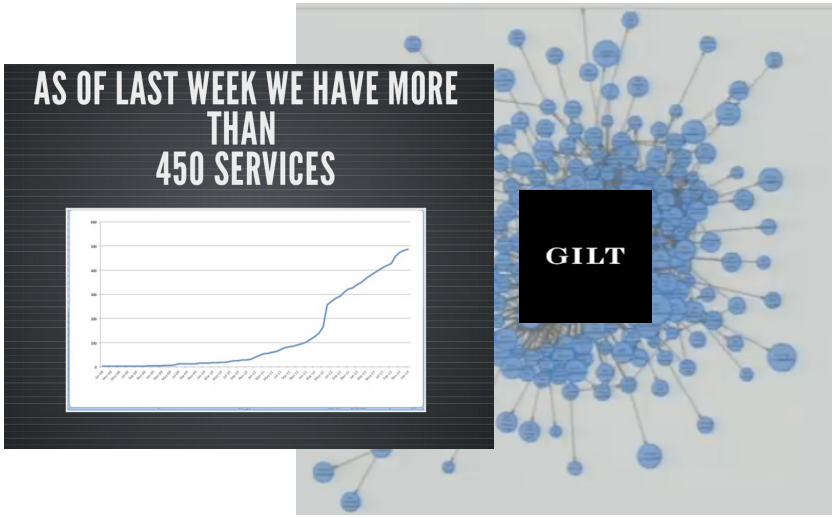


NETFLIX



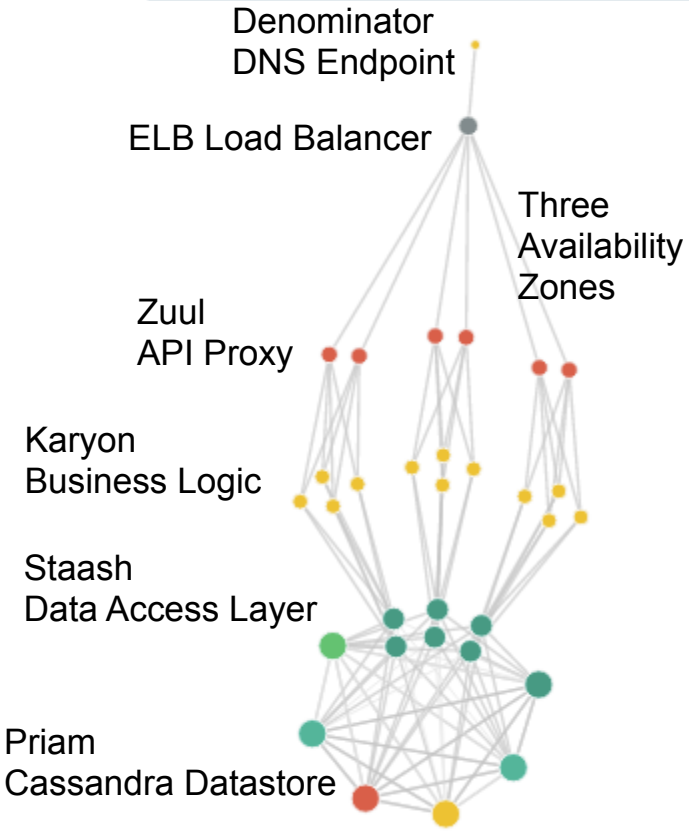
Interesting architectures have a lot of microservices! Flow visualization is a big challenge.





See http://www.slideshare.net/LappleApple/gilt-from-monolith-ruby-app-to-micro-service-scala-service-architecture

Simulated Microservices



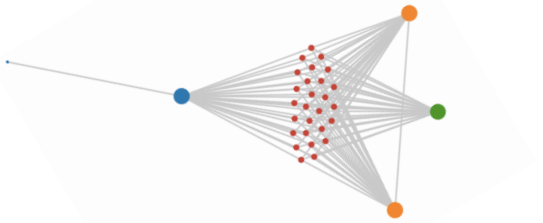
Model and visualize microservices
Simulate interesting architectures
Generate large scale configurations
Eventually stress test real tools

Code: github.com/adrianco/spigo
Simulate Protocol Interactions in Go
Visualize with D3
See for yourself: http://simianviz.surge.sh
Follow @simianviz for updates

Definition of an architecture

Header includes chaos monkey victim

See for yourself: http://simianviz.surge.sh/lamp



```
{
    "arch": "lamp",
    "description":"Simple LAMP stack",
    "version": "arch-0.0",
    "victim": "webserver",
    "services": [
        { "name": "rds-mysql",     "package": "store",       "count": 2,  "regions": 1, "dependencies": [] },
        { "name": "memcache",      "package": "store",       "count": 1,  "regions": 1, "dependencies": [] },
        { "name": "webserver",     "package": "monolith",    "count": 18, "regions": 1, "dependencies": ["memcache", "rds-mysql"] },
        { "name": "webserver-elb", "package": "elb",         "count": 0,  "regions": 1, "dependencies": ["webserver"] },
        { "name": "www",           "package": "denominator", "count": 0,  "regions": 0, "dependencies": ["webserver-elb"] }
    ]
}
```

New tier name

Tier package

Node count

0 = non Regional

List of tier dependencies

Running Spigo

-a architecture lamp
-j graph json/lamp.json
-d run for 2 seconds

```
$ ./spigo -a lamp -j -d 2
2016/01/26 23:04:05 Loading architecture from json_arch/lamp_arch.json
2016/01/26 23:04:05 lamp.edda: starting
2016/01/26 23:04:05 Architecture: lamp Simple LAMP stack
2016/01/26 23:04:05 architecture: scaling to 100%
2016/01/26 23:04:05 lamp.us-east-1.zoneB.eureka01....eureka.eureka: starting
2016/01/26 23:04:05 lamp.us-east-1.zoneA.eureka00....eureka.eureka: starting
2016/01/26 23:04:05 lamp.us-east-1.zoneC.eureka02....eureka.eureka: starting
2016/01/26 23:04:05 Starting: {rds-mysql     store 1 2 []}
2016/01/26 23:04:05 Starting: {memcache     store 1 1 []}
2016/01/26 23:04:05 Starting: {webserver     monolith 1 18 [memcache rds-mysql]}
2016/01/26 23:04:05 Starting: {webserver-elb     elb 1 0 [webserver]}
2016/01/26 23:04:05 Starting: {www     denominator 0 0 [webserver-elb]}
2016/01/26 23:04:05 lamp.*.*.www00....www.denominator activity rate  10ms
2016/01/26 23:04:06 chaosmonkey delete: lamp.us-east-1.zoneC.webserver02....webserver.monolith
2016/01/26 23:04:07 asgard: Shutdown
2016/01/26 23:04:07 lamp.us-east-1.zoneB.eureka01....eureka.eureka: closing
2016/01/26 23:04:07 lamp.us-east-1.zoneA.eureka00....eureka.eureka: closing
2016/01/26 23:04:07 lamp.us-east-1.zoneC.eureka02....eureka.eureka: closing
2016/01/26 23:04:07 spigo: complete
2016/01/26 23:04:07 lamp.edda: closing
```

Riak IoT Architecture

```
{
    "arch": "riak",
    "description":"Riak IoT ingestion example for the RICON 2015 presentation",
    "version": "arch-0.0",
    "victim": "",
    "services": [
        { "name": "riakTS",      "package": "riak",          "count": 6, "regions": 1, "dependencies": ["riakTS", "eureka"]},
        { "name": "ingester",    "package": "staash",        "count": 6, "regions": 1, "dependencies": ["riakTS"]},
        { "name": "ingestMQ",    "package": "karyon",        "count": 3, "regions": 1, "dependencies": ["ingester"]},
        { "name": "riakKV",      "package": "riak",          "count": 3, "regions": 1, "dependencies": ["riakKV"]},
        { "name": "enricher",    "package": "staash",        "count": 6, "regions": 1, "dependencies": ["riakKV", "ingestMQ"]},
        { "name": "enrichMQ",    "package": "karyon",        "count": 3, "regions": 1, "dependencies": ["enricher"]},
        { "name": "analytics",   "package": "karyon",        "count": 6, "regions": 1, "dependencies": ["ingester"]},
        { "name": "analytics-elb", "package": "elb",         "count": 0, "regions": 1, "dependencies": ["analytics"]},
        { "name": "analytics-api", "package": "denominator", "count": 0, "regions": 0, "dependencies": ["analytics-elb"]},
        { "name": "normalization", "package": "karyon",      "count": 6, "regions": 1, "dependencies": ["enrichMQ"]},
        { "name": "iot-elb",     "package": "elb",           "count": 0, "regions": 1, "dependencies": ["normalization"]},
        { "name": "iot-api",     "package": "denominator",   "count": 0, "regions": 0, "dependencies": ["iot-elb"]},
        { "name": "stream",      "package": "karyon",        "count": 6, "regions": 1, "dependencies": ["ingestMQ"]},
        { "name": "stream-elb", "package": "elb",            "count": 0, "regions": 1, "dependencies": ["stream"]},
        { "name": "stream-api", "package": "denominator",    "count": 0, "regions": 0, "dependencies": ["stream-elb"]}
    ]
}
```

New tier name

Tier package

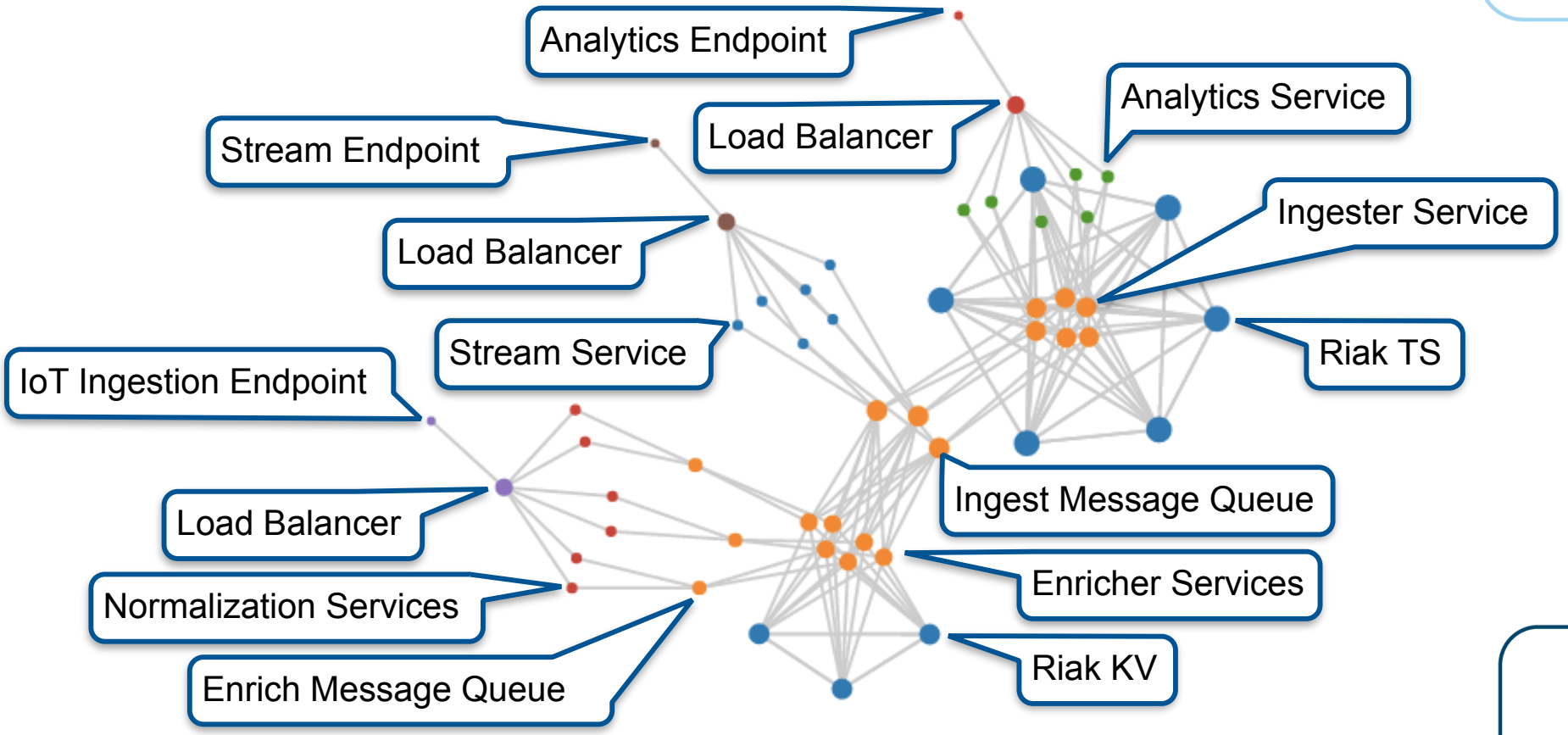
Node count

0 = non Regional

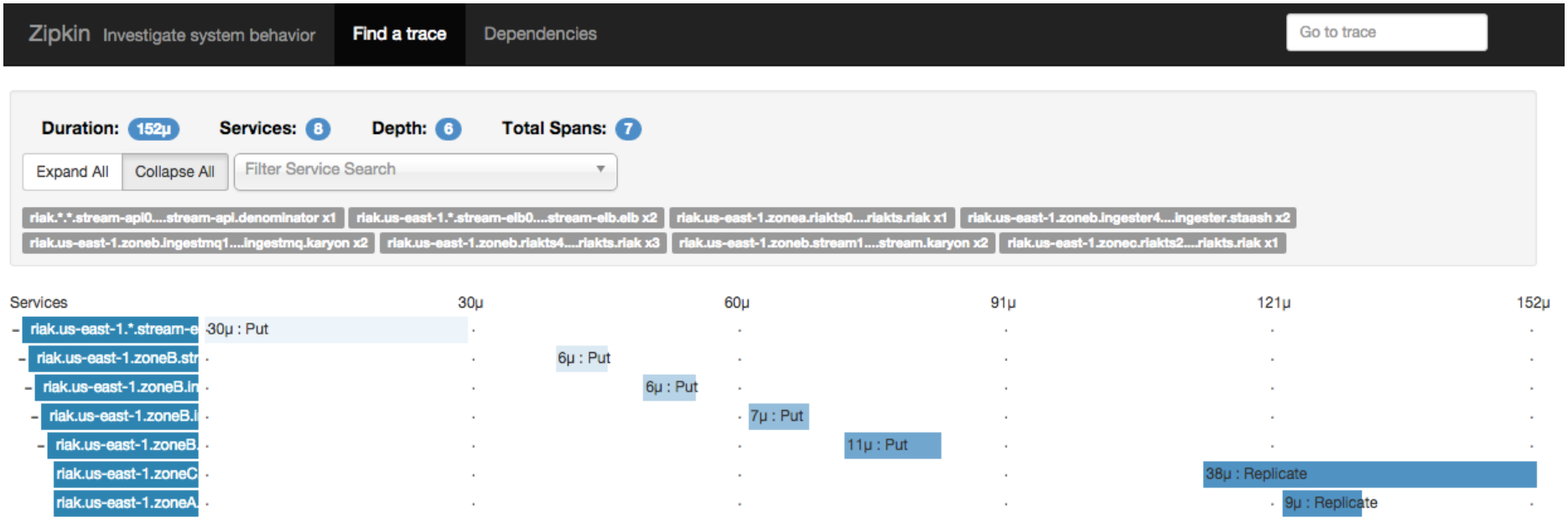
List of tier dependencies

Single Region Riak IoT

See for yourself: http://simianviz.surge.sh/riak

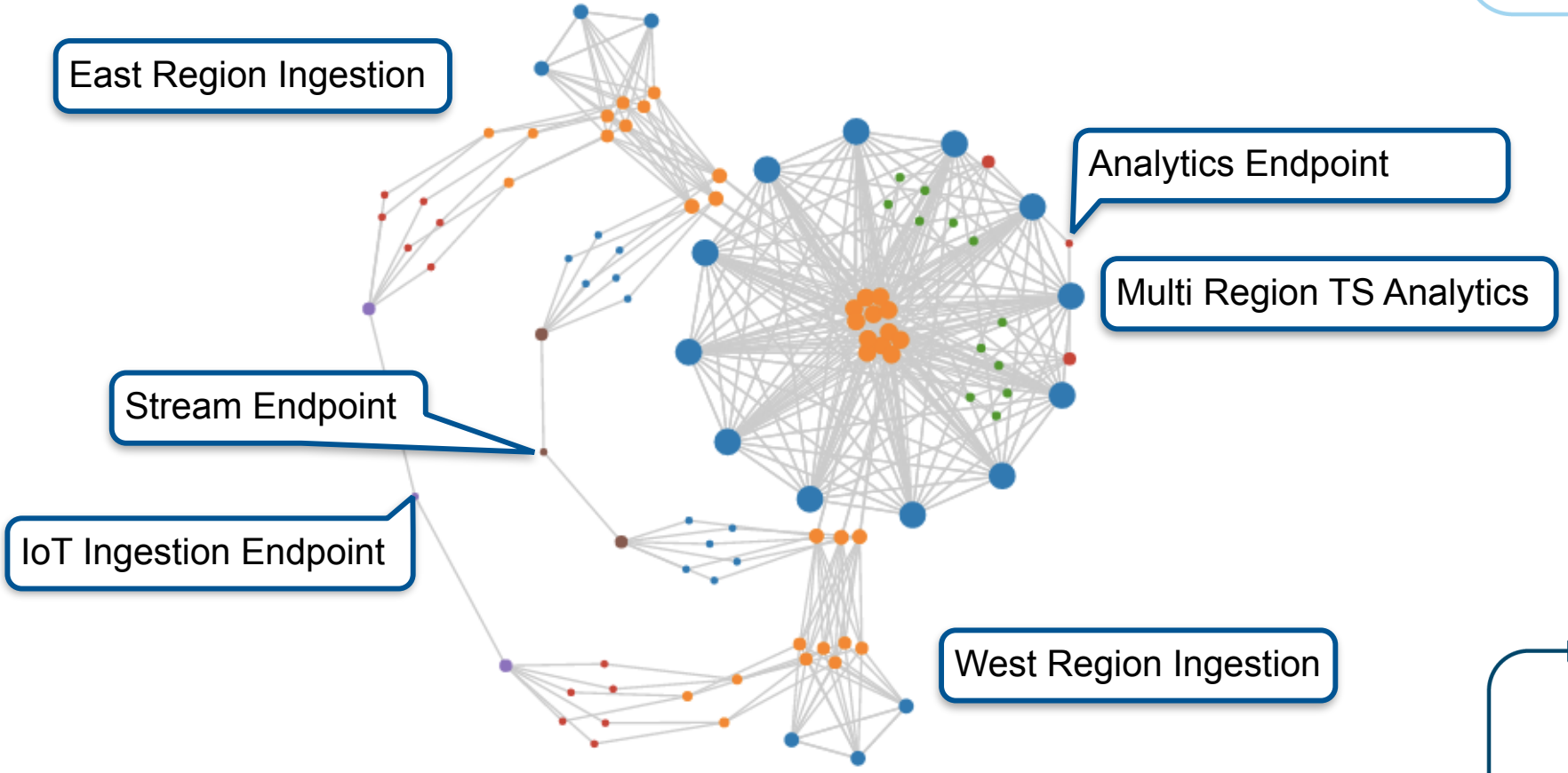


Trace for one Spigo Flow



Two Region Riak IoT

See for yourself: http://simianviz.surge.sh/riak

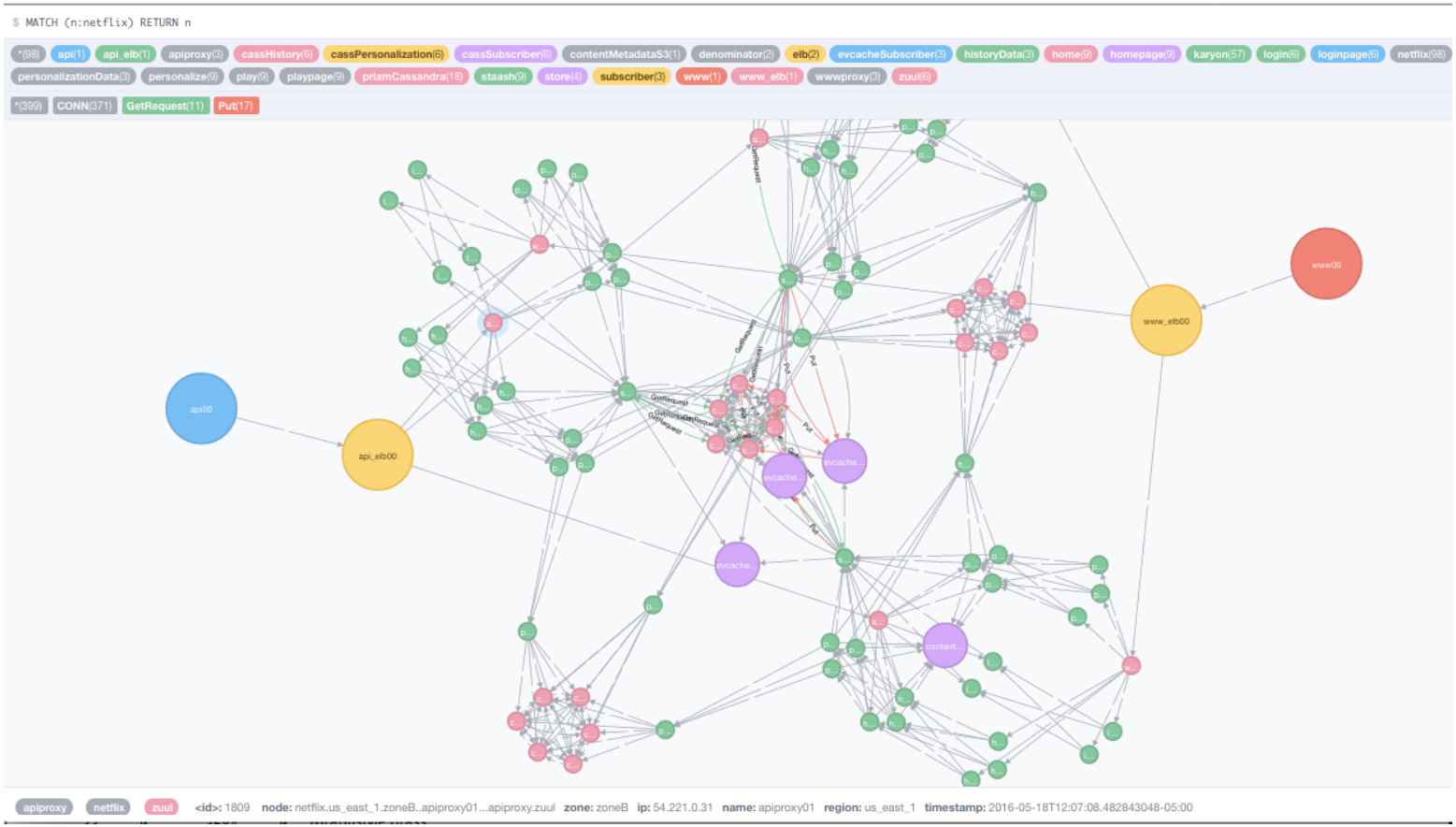


Spigo with Neo4j

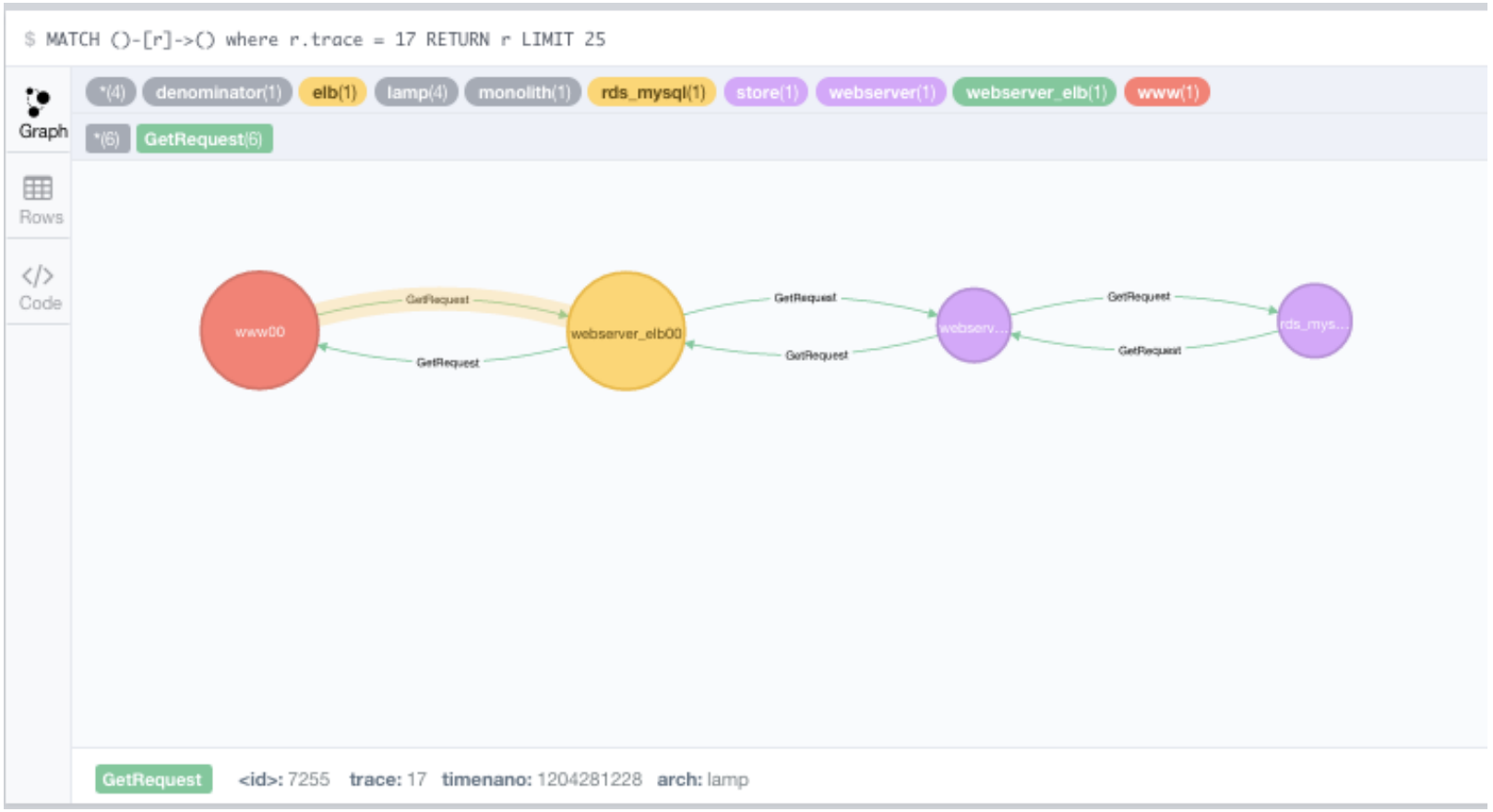
-a architecture netflix
-d run for 2 seconds
-n graph and flows written to Neo4j
-c flows json_metrics/netflix_flow.json
-kv chat:200ms start flows at 5/sec

```
$ ./spigo -a netflix -d 2 -n -c -kv chat:200ms
2016/05/18 12:07:08 Graph will be written to Neo4j via NEO4JURL=localhost:7474
2016/05/18 12:07:08 Loading architecture from json_arch/netflix_arch.json
2016/05/18 12:07:08 HTTP metrics now available at localhost:8123/debug/vars
2016/05/18 12:07:08 netflix.edda: starting
2016/05/18 12:07:08 Architecture: netflix A simplified Netflix service. See http://netflix.github.io/ to decode the package names
2016/05/18 12:07:08 architecture: scaling to 100%
2016/05/18 12:07:08 Starting: {cassSubscriber     priamCassandra 1 6 [cassSubscriber eureka]}
2016/05/18 12:07:08 netflix.us-east-1.zoneA..eureka00...eureka.eureka: starting
2016/05/18 12:07:08 netflix.us-east-1.zoneB..eureka01...eureka.eureka: starting
2016/05/18 12:07:08 netflix.us-east-1.zoneC..eureka02...eureka.eureka: starting
2016/05/18 12:07:08 Starting: {evcacheSubscriber     store 1 3 []}
2016/05/18 12:07:08 Starting: {subscriber     staash 1 3 [cassSubscriber evcacheSubscriber]}
2016/05/18 12:07:08 Starting: {cassPersonalization     priamCassandra 1 6 [cassPersonalization eureka]}
2016/05/18 12:07:08 Starting: {personalizationData     staash 1 3 [cassPersonalization]}
2016/05/18 12:07:08 Starting: {cassHistory     priamCassandra 1 6 [cassHistory eureka]}
2016/05/18 12:07:08 Starting: {historyData     staash 1 3 [cassHistory]}
2016/05/18 12:07:08 Starting: {contentMetadataS3     store 1 1 []}
2016/05/18 12:07:08 Starting: {personalize     karyon 1 9 [contentMetadataS3 subscriber historyData personalizationData]}
2016/05/18 12:07:08 Starting: {login     karyon 1 6 [subscriber]}
2016/05/18 12:07:08 Starting: {home     karyon 1 9 [contentMetadataS3 subscriber personalize]}
2016/05/18 12:07:08 Starting: {play     karyon 1 9 [contentMetadataS3 historyData subscriber]}
2016/05/18 12:07:08 Starting: {loginpage     karyon 1 6 [login]}
2016/05/18 12:07:08 Starting: {homepage     karyon 1 9 [home]}
2016/05/18 12:07:08 Starting: {playpage     karyon 1 9 [play]}
2016/05/18 12:07:08 Starting: {wwwproxy     zuul 1 3 [loginpage homepage playpage]}
2016/05/18 12:07:08 Starting: {apiproxy     zuul 1 3 [login home play]}
2016/05/18 12:07:08 Starting: {www-elb     elb 1 0 [wwwproxy]}
2016/05/18 12:07:08 Starting: {api-elb     elb 1 0 [apiproxy]}
2016/05/18 12:07:08 Starting: {www     denominator 0 0 [www-elb]}
2016/05/18 12:07:08 Starting: {api     denominator 0 0 [api-elb]}
2016/05/18 12:07:08 netflix.*.*..api00...api.denominator activity rate  200ms
2016/05/18 12:07:09 chaosmonkey delete: netflix.us-east-1.zoneA..homepage03...homepage.karyon
2016/05/18 12:07:10 asgard: Shutdown
2016/05/18 12:07:10 Saving 108 histograms for Guesstimate
2016/05/18 12:07:10 Saving 108 histograms for Guesstimate
2016/05/18 12:07:10 netflix.us-east-1.zoneC..eureka02...eureka.eureka: closing
2016/05/18 12:07:10 netflix.us-east-1.zoneA..eureka00...eureka.eureka: closing
2016/05/18 12:07:10 netflix.us-east-1.zoneB..eureka01...eureka.eureka: closing
2016/05/18 12:07:10 spigo: complete
```

Neo4j Visualization



Neo4j Trace Flow Queries



Conclusions

CSP is too limited
π-Calculus syntax is incomprehensible
Occam-Pi makes CSP and π-Calculus readable
Go concurrency syntax is clumsy in places but works
Showed some useful channel based Go-π idioms
Pass channels over channels for dynamic routing
Go works well for actor like simulation

Q&A

Adrian Cockcroft @adrianco
http://slideshare.com/adriancockcroft
http://github.com/adrianco/spigo