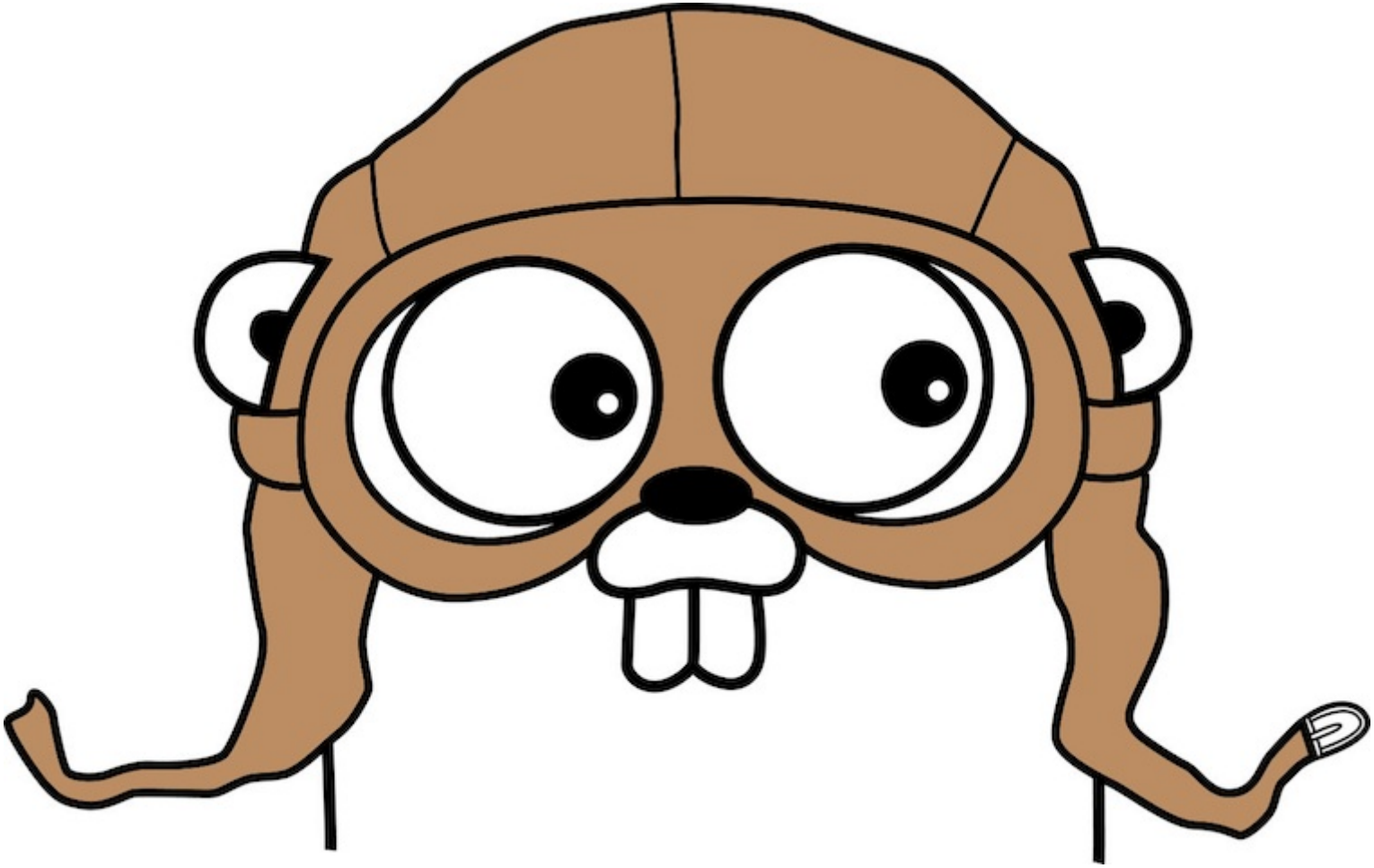
Composing Channels for Easy Concurrency, Part 2

October 21, 2015, Dallas, Texas, USA

John Scott
Consultant, American Messaging, 2006-now
Founder, SetSpace, Inc, 1998-now

The single biggest problem in communication is the illusion that it has taken place.

George Bernard Shaw



Share Memory by Communicating

Go is a Blend of Sequential and Concurrent Coding

The Big Threes of Systems Programming

- Modularity
- Composition
- Concurrency

Modular

No Side Effects, Code is Local

Composition

Create New Modules from Existing

Concurrency

Concurrency is the composition of independently executing processes.

Parallelism is the simultaneous execution of any processes.

Unix Shell Pipeline is Concurrency

```
find . -name '*.txt'  |  bzip2 >files.bz2
```

Find All text Files and Compress into files.bz2

```
ps -el  |  grep postgres  |  grep -v grep  |  mail -s 'Postgres Processes' jmscott@setspace.com
```

Mail List of All PostgreSQL Processes to jmscott@setspace.com

Background Processes Running in Parallel

```
slow_job1 &
slow_job2 &
slow_job3 &
```

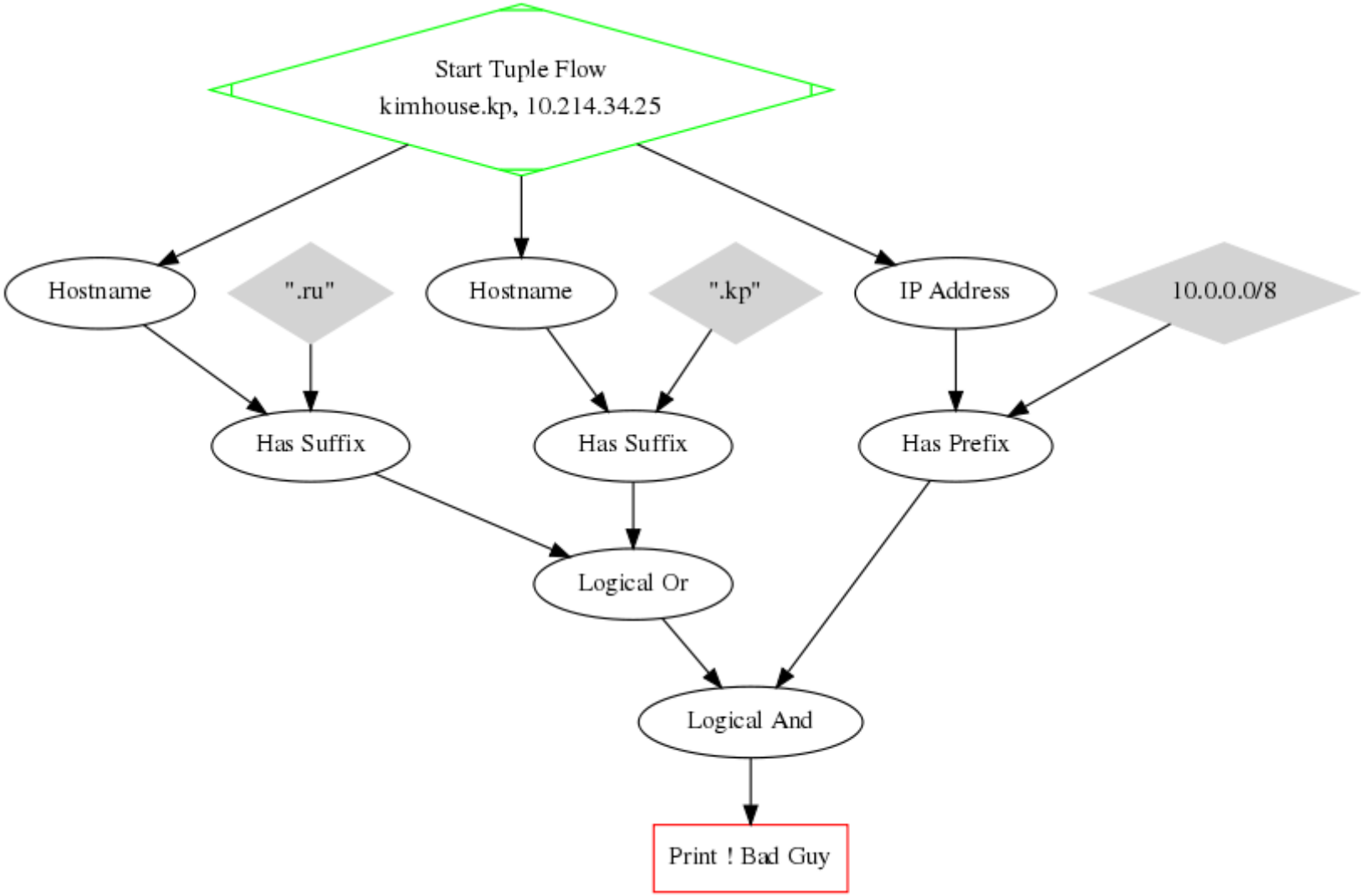
Not Concurrent Since Not Composed into Single Process

Cheap Broadcast by Closing Channel

Read on Closed Channel Always Returns Nil Or Zero

```
func pipe_element(in chan int) (out chan int) {

    out = make(chan int)

    go func() {
        defer close(out)

        for {
            request := <-in
            if request == 0 {
                return
            }

            //  ... do some work

            out <- 42
        }
     }()
     return out
}
```

Complex Flow Graph Descibes Query



Query: (hostname like "%ru" or hostname like "%kp") and ip4 like "10.%"

Flow Graph is Actually a Query Parse Tree

```
(hostname like "%ru" or hostname like "%kp") and ip4 like "10.%"
```

Simple Six Instructions for "Query" Language

- Logical AND
- Logical OR
- Has a Prefix String
- Has a Suffix String
- Project (Push) Tuple Attribute (Column)
- Print

How to Coordinate Many Go Routines

- Close a Single Broadcast Channel on Which Many GoRoutines Listen
- Send Next Request for Work to Single GoRoutine as Channel Over Channel

Confluence - an act or process of merging.

Flow Data Structures

Flow Coordinates Many Query Predicates Analyzing a Single Tuple

```
type flow struct {

    //  all expressions are resolved when closed
    resolved chan struct{}

    //  request next flow
    next chan chan *flow

    //  count total unresolved query predicate expressions
    confluent_count uint32

    //  tuple being analyzed: [hostname, ip4]
    tuple map[string]string
}

//  A is always resolved and B is "conflueing"
var flowA, flowB *flow
```

Create a New Flow

```
func (flo *flow) new(inbound map[string]string) *flow {

    return &flow{
        resolved: make(chan struct{}),
        next:     make(chan chan *flow),
        tuple:    inbound,
    }
}
```

Initialize the 0'th Flow and Victious Attack Data

```
var attack [4]map[string]string

func init() {
    flowA = flowA.new(nil)
    close(flowA.resolved) //  0'th already resolved

    attack[0] = map[string]string{
        "ip4":      "173.193.106.242",
        "hostname": "condor.setspace.com",
    }
    attack[1] = map[string]string{
        "ip4":      "23.207.24.80",
        "hostname": "www.apple.com",
    }
    attack[2] = map[string]string{
        "ip4":      "10.214.34.11",
        "hostname": "kimroom.kp",
    }
    attack[3] = map[string]string{
        "ip4":      "10.214.34.25",
        "hostname": "theputins.ru",
    }
}
```

Coordinate Many Query Predicates

```
func (flo *flow) get() *flow {

    <-flo.resolved

    reply := make(chan *flow)

    flo.next <- reply

    return <-reply
}
```

Each Predicate GoRoutine Calls flow.get()

"Instruction" to Match Suffix of String

```
func (flo *flow) has_suffix(
    suffix_constant string,
    in chan string,
) (out chan bool) {

    out = make(chan bool)

    go func() {

        for flo = flo.get(); flo != nil; flo = flo.get() {

            out <- HasSuffix(<-in, suffix_constant)
        }
    }()

    return out
}
```

"Instruction" to Push Attribute of Input Tuple

```
func (flo *flow) project_tuple(att_name string) (out chan string) {

    out = make(chan string)

    go func() {

        for flo = flo.get(); flo != nil; flo = flo.get() {

            out <- flo.tuple[att_name]
        }
    }()

    return out
}
```

"Instruction" to Print Answer of Flow Over Tuple

```
func (flo *flow) print(in chan bool) (out chan struct{}) {

    out = make(chan struct{})
    go func() {
        defer close(out)

        for flo = flo.get(); flo != nil; flo = flo.get() {

            who := "+ good guy"
            if <-in {
                who = "! bad guy"
            }

            Println(who, "->", flo.tuple)

            out <- struct{}{} //  why the extra {}?
        }
    }()

    return out
}
```

"Conflue" has Resolved flowA to Start flowB

```
func conflue(out chan struct{}) {

    //  wait for all expressions in flowA to resolve
    for flowA.confluent_count > 0 {

        reply := <-flowA.next
        flowA.confluent_count--

        reply <- flowB
        flowB.confluent_count++
    }

    //  wait for flowB to finish with final print statement
    <-out

    //  start moving flowB to next flow
    close(flowB.resolved)

    //  and the wheel turns ...
    flowA = flowB
}
```

Simple Example Confluence

```
func simple_conflue(source map[string]string) {

    //  compile expression
    //
    //    print(hostname like "%.kp")

    out :=
        flowA.print(
            flowA.has_suffix(".kp",
                flowA.project_tuple("hostname")))

    //  three go routines will converge
    flowA.confluent_count = 3

    flowB = flowB.new(source)
    conflue(out)
}
```

Run Simple Example Query

```
func simple() {

    trivial := map[string]string{
        "hostname": "kimsroom.kp",
    }

    simple_conflue(trivial)
}

func main() {

    simple()
}
```

Run

"Instruction" to Logical AND

```
func (flo *flow) logical_AND(left, right chan bool) (out chan bool) {

    out = make(chan bool)

    go func() {
        defer close(out)

        for flo = flo.get(); flo != nil; flo = flo.get() {

            l, r := false, false
            for count := 0; count < 2; count++ {
                select {
                case l = <-left:
                case r = <-right:
                }
            }

            out <- l && r
        }
    }()
    return out
}
```

"Instruction" to Logical OR

```
func (flo *flow) logical_OR(left, right chan bool) (out chan bool) {

    out = make(chan bool)

    go func() {
        defer close(out)

        for flo = flo.get(); flo != nil; flo = flo.get() {

            l, r := false, false
            for count := 0; count < 2; count++ {
                select {
                case l = <-left:
                case r = <-right:
                }
            }

            out <- l || r
        }
    }()
    return out
}
```

"Instruction" to Match Prefix of String

```
func (flo *flow) has_prefix(
    prefix_constant string,
    in chan string,
) (out chan bool) {

    out = make(chan bool)

    go func() {

        for flo = flo.get(); flo != nil; flo = flo.get() {

            out <- HasPrefix(<-in, prefix_constant)
        }
    }()

    return out
}
```

Compile Complex "Attack" Query

```
func compile() (out chan struct{}) {
    out = flowA.print(
        flowA.logical_AND(

            //  ip4 like "10.%"
            flowA.has_prefix(
                "10.",
                flowA.project_tuple("ip4"),
            ),

            flowA.logical_OR(

                // hostname like "%.ru"
                flowA.has_suffix(
                    ".ru",
                    flowA.project_tuple("hostname"),
                ),

                // hostname like "%.kp"
                flowA.has_suffix(
                    ".kp",
                    flowA.project_tuple("hostname")))))
    flowA.confluent_count = 9
    return
}
```

Attacking!!!!

```
func attacking() {

    out := compile()
    for _, a := range attack {
        flowB = flowB.new(a)
        conflue(out)
    }
}
```

```
func main() {

    simple()
}
```

Run

Thank you

October 21, 2015, Dallas, Texas, USA

John Scott
Consultant, American Messaging, 2006-now
Founder, SetSpace, Inc, 1998-now

jmscott@setspace.com (mailto:jmscott@setspace.com)
john.scott@americanmessaging.net (mailto:john.scott@americanmessaging.net)