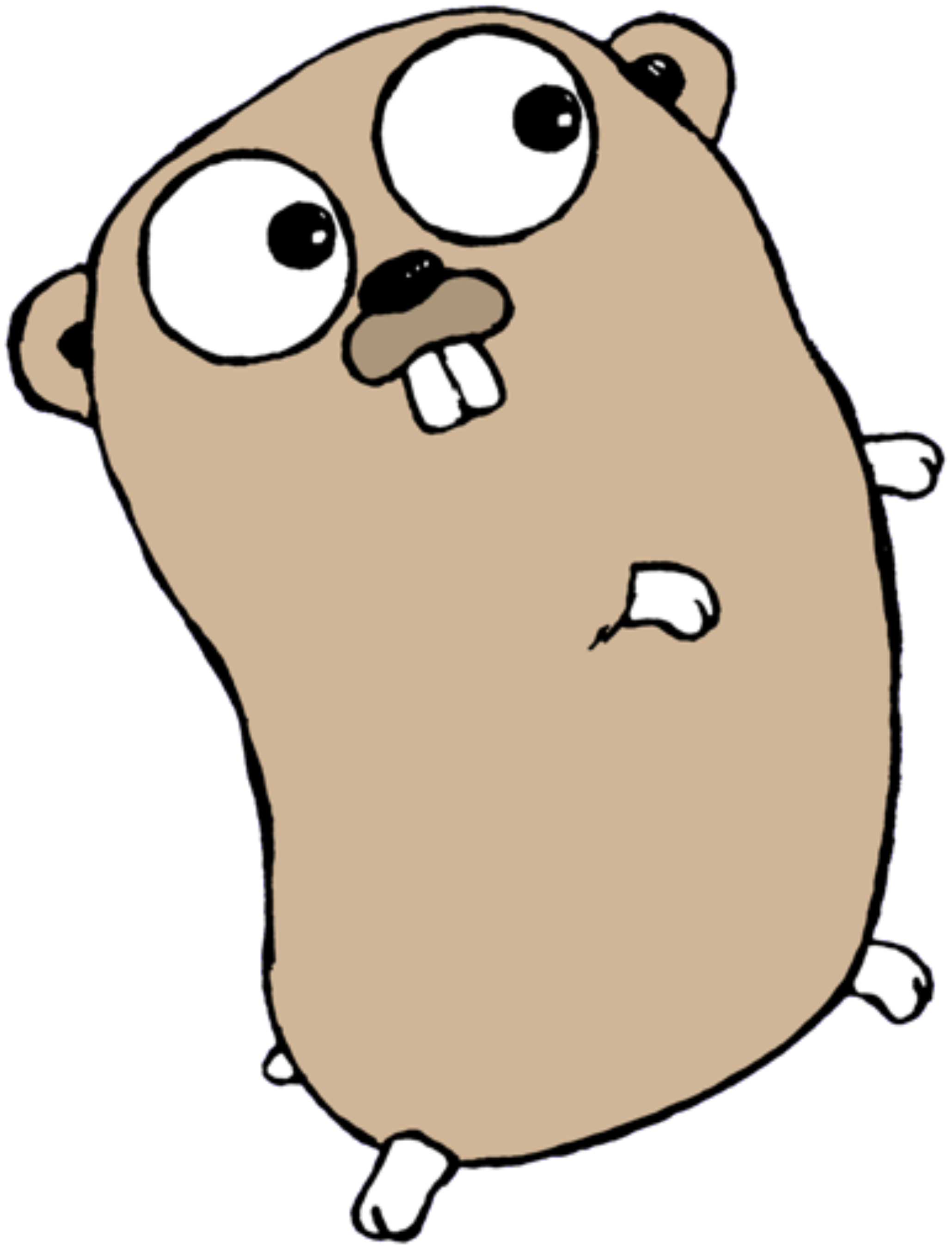
11 IO

BootCamp

Real World Go

Andrew Gerrand
May 9, 2011



Background

Why Go?

- Statically typed languages are efficient, but typically bureaucratic and overly complex.
- Dynamic languages can be easy to use, but are error-prone, inefficient, and break down at scale.
- Concurrent programming is hard (threads, locks, headache).

- “Speed, reliability, or simplicity: pick two.” (sometimes just one)
- Can’t we do better?

What is Go?

- Go is a modern, general purpose language.

- Compiles to native machine code (32-bit and 64-bit x86, ARM).
- Statically typed.
- Lightweight syntax.
- Simple type system.
- Concurrency primitives.

Tenets of Go's design

- Simplicity
  - Each language feature should be easy to understand.
- Orthogonality
  - Go's features should interact in predictable and consistent ways.
- Readability
  - What is written on the page should be comprehensible with little context.

"Consensus drove the design. Nothing went into the language until [Ken Thompson, Robert Griesemer, and myself] all agreed that it was right. Some features didn't get resolved until after a year or more of discussion."

Rob Pike

Hello, world

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

Hello, world 2.0

Serving “Hello, world” at `http://localhost:8080/world`

```
package main

import (
    "fmt"
    "http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello, "+r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

A simple type system

Simple type system

Go is statically typed, but type inference saves repetition.

Java:

```
Integer i = new Integer(1);
```

C/C++:

```
int i = 1;
```

Go:

```
i := 1 // type int
pi := 3.142 // type float64
greeting := "Hello, Bootcamp!" // type string
mul := func(x, y int) int { return x * y }
       // type func(int, int) int
```

Types and methods

You can define methods on any type:

```
type Point struct {
    X, Y float64
}

func (p Point) Abs() float64 {
    return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

p := Point{4, 3} // type Point
p.Abs() // == 5.0
```

Types and methods

You can define methods on *any* type:

```
type MyFloat float64

func (m MyFloat) Abs() float64 {
    f := float64(m)
    if f < 0 {
        return -f
    }
    return f
}

f := MyFloat(-42)
f.Abs() // == 42.0
```

Go “objects” are just values. There is no “box”.

Interfaces

Interfaces specify behaviors.

An interface type defines a set of methods:

```
type Abser interface {
    Abs() float64
}
```

A type that implements those methods implements the interface:

```
func PrintAbs(a Abser) {
    fmt.Printf("Absolute value: %.2f\n", a.Abs())
}
```

```
PrintAbs(MyFloat(-10))
```

```
PrintAbs(Point{3, 4})
```

Types implement interfaces implicitly.

There is no “implements” declaration.

Interfaces in practice

From the `io` package in the standard library:

```
type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
```

There are many Writer implementations throughout the standard library and other Go code.

We've already seen an example:

```
func handler(w http.ResponseWriter, r ...) {
    fmt.Fprint(w, "Hello, "+r.URL.Path[1:])
}
```

The `fmt.Fprint` function takes an `io.Writer` as its first argument, and `http.ResponseWriter` implements the `Write` method.

The `fmt` package doesn't know `http`. It just works.

Concurrency

Concurrency

In UNIX we think about **processes** connected by **pipes**:

```
find ~/go/src/pkg | grep _test.go$ | xargs wc -l
```

Each tool designed to do one thing and to do it well.

The Go analogue: **goroutines** connected by **channels**.

Concurrency: goroutines

Goroutines are like threads:

- They share memory.

But cheaper:

- Smaller, segmented stacks.
- Many goroutines per operating system thread.

Start a new goroutine with the go keyword:

```
i := pivot(s)
go sort(s[:i])
go sort(s[i:])
```

Concurrency: channels

Channels are a typed conduit for:

- Synchronization.
- Communication.

The channel operator `<-` is used to send and receive values:

```
func compute(ch chan int) {
    ch <- someComputation()
}

func main() {
    ch := make(chan int)
    go compute(ch)
    result := <-ch
}
```

Concurrency: synchronization

Look back at the sort example - how to tell when it's done?

Use a channel to synchronize goroutines:

```
done := make(chan bool)
doSort := func(s []int) {
    sort(s)
    done <- true
}
i := pivot(s)
go doSort(s[:i])
go doSort(s[i:])
<-done
<-done
```

Unbuffered channel operations are synchronous;
the send/receive happens only when both sides are ready.

Concurrency: communication

A common task: many workers feeding from task pool.

Traditionally, worker threads contend over a lock for work:

```
type Task struct {
    // some state
}
type Pool struct {
    Mu    sync.Mutex
    Tasks []Task
}
func worker(pool *Pool) { // many of these run concurrently
    for {
        pool.Mu.Lock()
        task := pool.Tasks[0]
        pool.Tasks = pool.Tasks[1:]
        pool.mu.Unlock()
        process(task)
    }
}
```

Concurrency: communication

A Go idiom: many worker goroutines receive tasks from a channel.

```
type Task struct {
    // some state
}
func worker(in, out chan *Task) {
    for {
        t := <-in
        process(t)
        out <- t
    }
}
func main() {
    pending, done := make(chan *Task), make(chan *Task)
    go sendWork(pending)
    for i := 0; i < 10; i++ {
        go worker(pending, done)
    }
    consumeWork(done)
}
```

Concurrency: philosophy

- Goroutines give the efficiency of an asynchronous model, but you can write code in a synchronous style.
- Easier to reason about: write goroutines that do their specific jobs well, and connect them with channels.
  - In practice, this yields simpler and more maintainable code.
- Think about the concurrency issues that matter:

“Don’t communicate by sharing memory.
Instead, share memory by communicating.”

Rich library support

- Diverse, carefully-constructed, consistent standard library
    - More than 150 packages
    - Constantly under development; improving every day
- Many great external libraries, too

  `http://godashboard.appspot.com/package` lists >200 packages
    - MySQL, MongoDB, and SQLite3 database drivers,
    - SDL bindings,
    - Protocol Buffers,
    - OAuth libraries,
    - and much more.

Go: What is it good for?

- Initially called it a “systems language.”
- People found this confusing (oops).
- Unexpected interest from users of scripting languages.
  - Attracted by an easy, reliable language that performs well.
- Diverse uses across the community:
  - scientific computing,
  - web applications,
  - graphics and sound,
  - network tools,
  - and much more.
- Now we call Go a “general-purpose language.”

Real World Go

Heroku

Who are Heroku?

- http://www.heroku.com/
- Heroku provides cloud hosting for Ruby programmers.
- Keith Rarick and Blake Mizerany were designing a “distributed init system” for
  - managing processes across clusters of machines, and
  - recovering gracefully from instance failures and network partitions.
- They need to reliably synchronize and share information among many servers.
- That’s why they wrote Doozer.

Heroku's Doozer

What is Doozer?

Doozer is a rock-solid basis for building distributed systems.

- A highly available (works during network partitions),
- consistent (no inconsistent writes),
- data store (for small amounts of data).

It provides a single fundamental synchronization primitive: *compare-and-set*.

Heroku's Doozer

What is it good for?

"Doozer is where you put the family jewels."

Example use cases:

- Database master election
- Name service
- Configuration

Heroku's Doozer

Why choose Go?

Go's concurrency primitives suit the problem:

- Doozer uses Paxos to achieve consensus between nodes,
- Paxos is a distributed algorithm described in terms of separate entities exchanging messages asynchronously,
- It is notoriously difficult to get right, but
- Goroutines and channels made it manageable.

"In the same way that garbage collectors improve upon malloc and free, we found that goroutines and channels improve upon the lock-based approach to concurrency. These tools let us avoid complex bookkeeping and stay focused on the problem at hand. We are still amazed at how few lines of code it took to achieve something renowned for being difficult."

- Blake Mizerany, The Go Programming Language Blog, April 2011

Heroku's Doozer

Why choose Go?

The convenience of the standard library:

> *"Using the* `websocket` *package, Keith was able to add the web viewer on his train ride home and without requiring external dependencies. This is a real testament to how well Go mixes systems and application programming."*

Mechanical source formatting settled arguments:

> *"One of our favorite productivity gains was provided by Go's source formatter:* `gofmt`*. We never argued over where to put a curly-brace, tabs vs. spaces, or if we should align assignments. We simply agreed that the buck stopped at the default output from gofmt. "*

MROffice

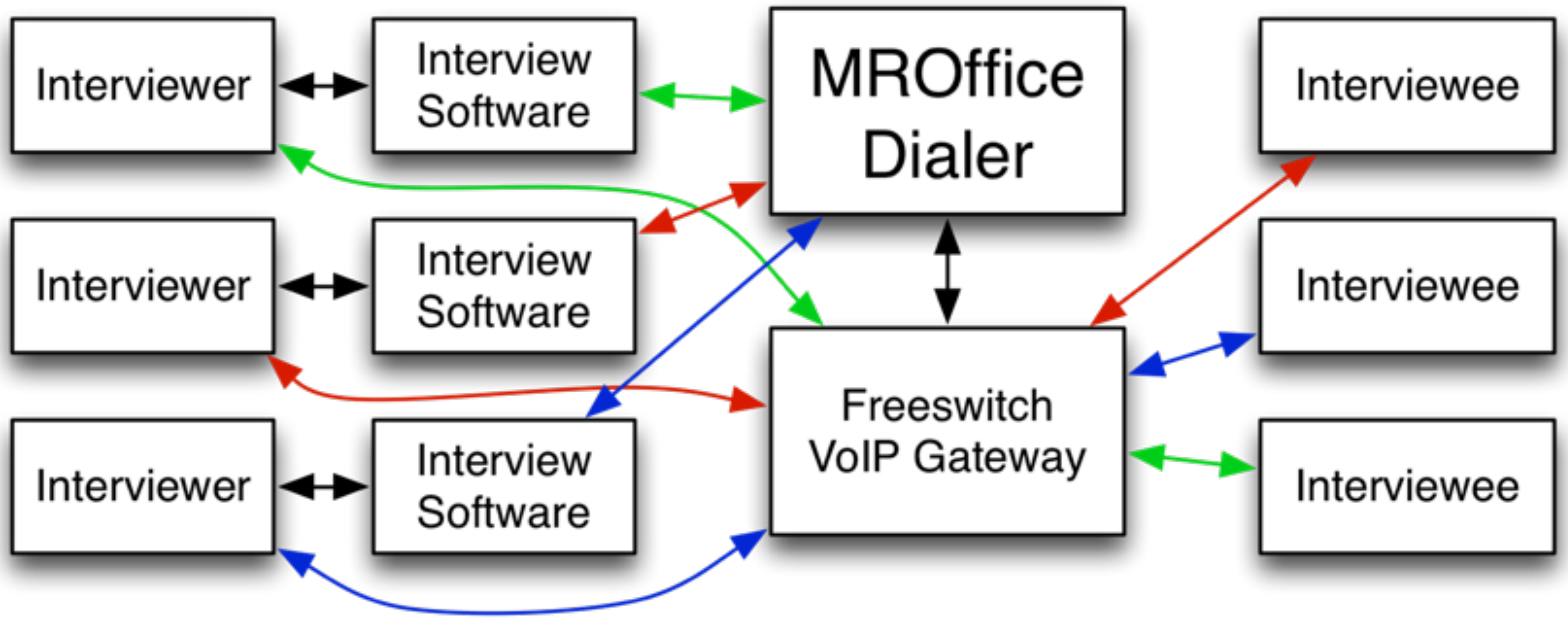
Who are MROffice?

- http://www.mroffice.org/
- Kees Varekamp; one man based in New Zealand.
- Has a background in market research software.
- Found most existing software in that space to be pretty bad.
- Launched MROffice in 2010 to provide better software to the Market Research industry.
- His flagship product is called Dialer.

MROffice Dialer

What does Dialer do?

- Connects interviewers in a call center to interviewees.
- A bridge between interview platforms (that provide scripts and collect statistics) and a VoIP dialer (to do the actual telephony).



MROffice Dialer

Switching languages

- Originally written in Python:

  "I LOVE Python; I use it for everything. But I found that for long-running server processes it might not be such a good choice: lots of runtime errors that could have been caught during compile time."

  "When Go came out it immediately made sense to me:
  Type safe, compiled, feels like a scripting language."

  - Kees at the Sydney Go User Group, March 2011

- So he ported the Python code to Go.

MROffice Dialer

- Why Go works
  - The concurrency model suited the problem. A goroutine to handle each call, interviewer, and interviewee, all communicating via channels.
  - The http and websocket libraries made it easy to write a management UI.
- Onward and upward
  - Beta product now running in multiple call centers.
  - Predictive dialer design that uses neural networks.
- Conclusions about Go
  - *“Excellent tutorials and documentation.”*
  - *“I’ve been converted to statically typed languages.”*
  - *“[Go is a] good compromise for cooperation between type purists and lazy scripters.”*

Atlassian

Atlassian make development and collaboration tools for software developers. They are mostly a Java shop.

They have a testing cluster of virtual machines, run over a large number of diskless hosts.

Its provisioning and monitoring system is written in Go.

The system is in three parts:

- Agent processes that run on each server, broadcasting the state of their VMs.
- A manager process that listens to the agent's broadcasts and takes action if a VM fails to report in.
- A command-line tool for issuing commands to the manager process ("shut down", "restart", etc.).

Go at Atlassian

The Agent

- A trivial Go program.
- It does three things, forever:
  - Read VM state from `/proc`,
  - Encode the state information as a protocol buffer,
  - Broadcast the protocol buffer via UDP.

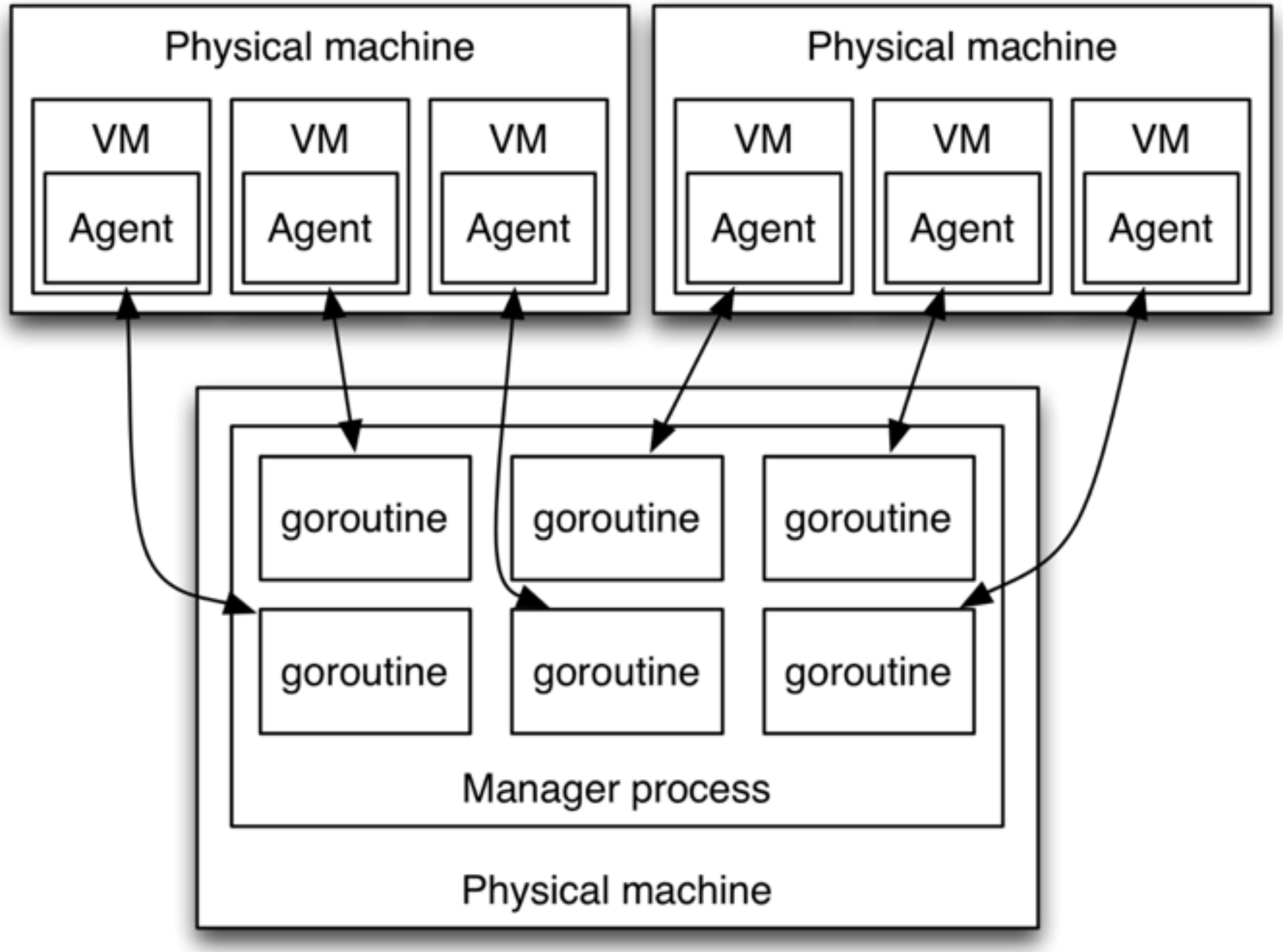
Go at Atlassian

The Manager

- Read a configuration file.
- Launch one goroutine for each VM in the cluster.
- Each goroutine
  - listens for announcements from its corresponding VM, and
  - issues instructions (shell commands) to keep it in the correct state.

Go at Atlassian

System overview



Go at Atlassian

Why Go works

- Easy to deploy; ship binaries with no dependencies:
  - *"The agent process runs on machines that netboot and run completely from RAM. A single static binary was a major saving, compared to a JVM or Python runtime."*

    - Dave Cheney, Atlassian Engineer
- "One goroutine per VM" maps nicely to their problem:
  - *"[This] is trivial in Go, but painful in [other languages]."*

Camlistore

What is Camlistore?

- http://www.camlistore.org/
- Brad Fitzpatrick wanted to store his data across all his devices, and to share that data with friends and the public.
- Camlistore is:
  - a content-addressable data store,
  - a synchronization and access-control mechanism,
  - an API,
  - a user interface,
  - your "home directory for the web",
  - programming language-agnostic.
- The largest parts of it are written in Go.

Camlistore

Use cases

- Personal backups, automatically synced to remote servers.
- Dropbox-style file synchronization across machines.
- Photo management and sharing.
- Web site content management.

Camlistore

The Go parts

- camlistored:
  - blobserver, a place to keep blocks of immutable data,
  - HTTP server for interacting with Camlistore clients,
  - HTTP user interface for users and administrators.
- command-line tools:
  - camput, put things in the blob store,
  - camget, get things from the blob store,
  - camsync, synchronize blob stores,
  - cammount, a FUSE filesystem for accessing your data.
- two dozen assorted libraries.

Camlistore

Why Go?

"I've been writing [Go] for over a year now and it makes me giddy about programming again.

Annoying things aren't annoying, trade-offs that I'd normally worry about I no longer worry about.

I bust out lots of fast, correct, maintainable testable code in very small amounts of time, without much code.

I haven't been this excited about a language in ages."

"We wrote [cammount] in two or three beers."

"I had the idea for Camlistore a long time ago, but before I learned Go it had always seemed too painful."

Go’s diverse talents

- Heroku: Doozer, a highly available consistent data store,
  - rock-solid systems programming.
- MROffice: a dialer for call centers,
  - simple, reliable applications programming.
- Atlassian: virtual machine cluster management,
  - utility programming with concurrency support.
- Camlistore: content addressable storage system,
  - “full stack” programming, from the data store to the UI.

Go is Open Source

- Development began at Google in 2007 as a 20% project.
- Released under a BSD-style license in November 2009.
- Since its release, more than 130 non-Google contributors have submitted over 1000 changes to the Go core.
- ~10 Google employees work on Go full-time.
- Two non-Google committers, dozens of regular contributors.
- All Go development (code reviews, etc.) takes place on a public mailing list.

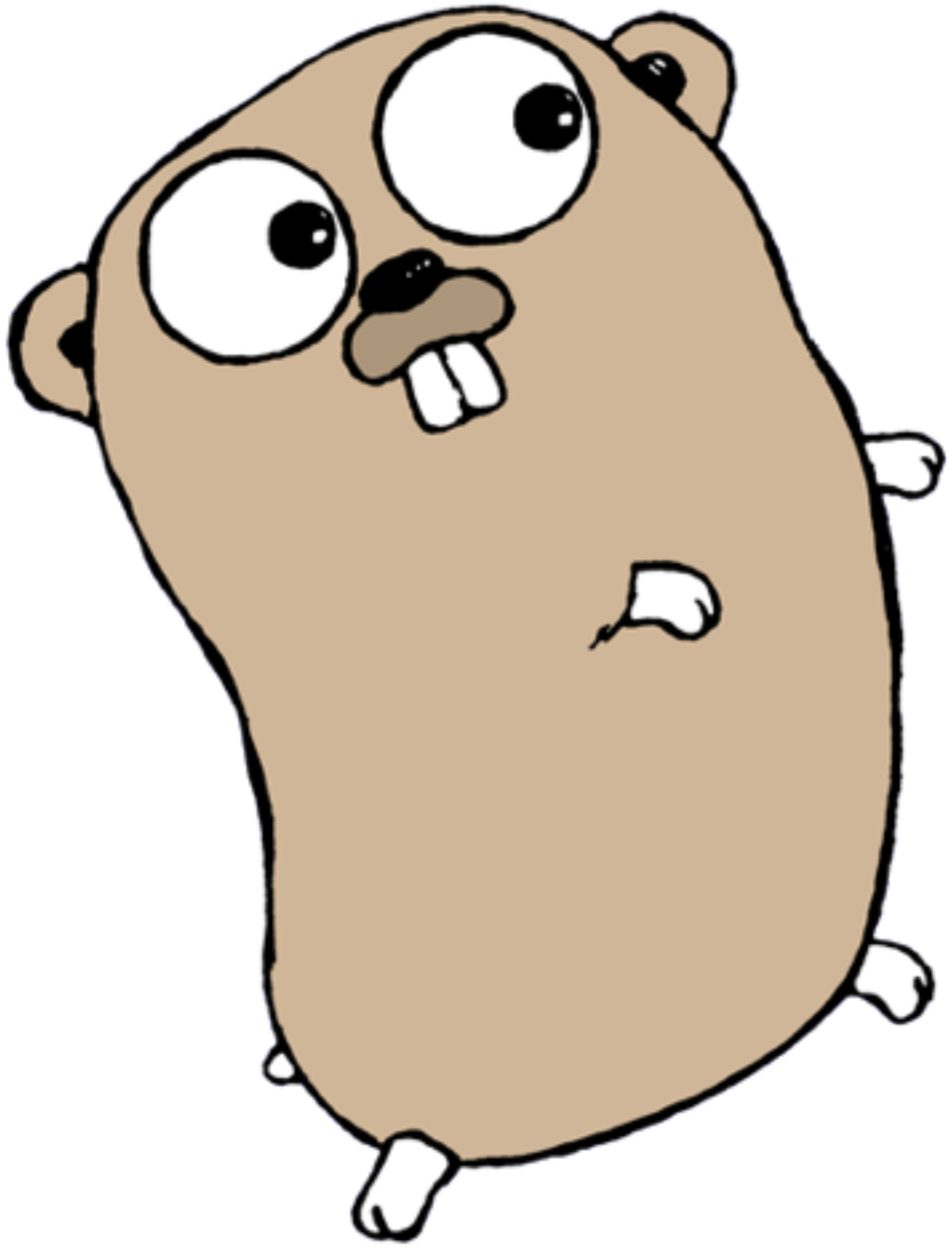
- You are invited to be involved!

More about Go

- You can learn more about Go at I/O:
  - "Get Started With Go" workshops at Bootcamp this afternoon,
  - "Writing Web Apps in Go" talk, 11:30am Tuesday,
  - Office hours: meet the Go team, 12-3pm both days.

- Also visit our web site and the official blog:
  - `http://golang.org/`
  - `http://blog.golang.org/`

Questions?



11 IO

BootCamp