Go Concurrency Patterns

Rob Pike
Google

Video

This talk was presented at Google I/O in June 2012.

Watch the talk on YouTube (http://www.youtube.com/watch?v=f6kdp27TYZs) 2

Introduction

3

Concurrency features in Go

People seemed fascinated by the concurrency features of Go when the language was first announced.

Questions:

- Why is concurrency supported?
- What is concurrency, anyway?
- Where does the idea come from?
- What is it good for?
- How do I use it?

Why?

Look around you. What do you see?

Do you see a single-stepping world doing one thing at a time?

Or do you see a complex world of interacting, independently behaving pieces?

That's why. Sequential processing on its own does not model the world's behavior.

What is concurrency?

Concurrency is the composition of independently executing computations.

Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world.

It is not parallelism.

6

Concurrency is not parallelism

Concurrency is not parallelism, although it enables parallelism.

If you have only one processor, your program can still be concurrent but it cannot be parallel.

On the other hand, a well-written concurrent program might run efficiently in parallel on a multiprocessor. That property could be important...

For more on that distinction, see the link below. Too much to discuss here.

golang.org/s/concurrency-is-not-parallelism (http://golang.org/s/concurrency-is-not-parallelism) 7

A model for software construction

Easy to understand.

Easy to use.

Easy to reason about.

You don't need to be an expert!

(Much nicer than dealing with the minutiae of parallelism (threads, semaphores, locks, barriers, etc.))

History

To many, the concurrency features of Go seemed new.

But they are rooted in a long history, reaching back to Hoare's CSP in 1978 and even Dijkstra's guarded commands (1975).

Languages with similar features:

- Occam (May, 1983)
- Erlang (Armstrong, 1986)
- Newsqueak (Pike, 1988)
- Concurrent ML (Reppy, 1993)
- Alef (Winterbottom, 1995)
- Limbo (Dorward, Pike, Winterbottom, 1996).

9

Distinction

Go is the latest on the Newsqueak-Alef-Limbo branch, distinguished by first-class channels.

Erlang is closer to the original CSP, where you communicate to a process by name rather than over a channel.

The models are equivalent but express things differently.

Rough analogy: writing to a file by name (process, Erlang) vs. writing to a file descriptor (channel, Go).

Basic Examples

11

A boring function

We need an example to show the interesting properties of the concurrency primitives.

To avoid distraction, we make it a boring example.

Run

12

Slightly less boring

Make the intervals between messages unpredictable (still under a second).

Run

13

Running it

The boring function runs on forever, like a boring party guest.

Run

14

Ignoring it

The go statement runs the function as usual, but doesn't make the caller wait.

It launches a goroutine.

The functionality is analogous to the & on the end of a shell command.

Run

15

Ignoring it a little less

When main returns, the program exits and takes the boring function down with it.

We can hang around a little, and on the way show that both main and the launched goroutine are running.

Run

16

Goroutines

What is a goroutine? It's an independently executing function, launched by a go statement.

It has its own call stack, which grows and shrinks as required.

It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.

It's not a thread.

There might be only one thread in a program with thousands of goroutines.

Instead, goroutines are multiplexed dynamically onto threads as needed to keep all the goroutines running.

But if you think of it as a very cheap thread, you won't be far off.

Communication

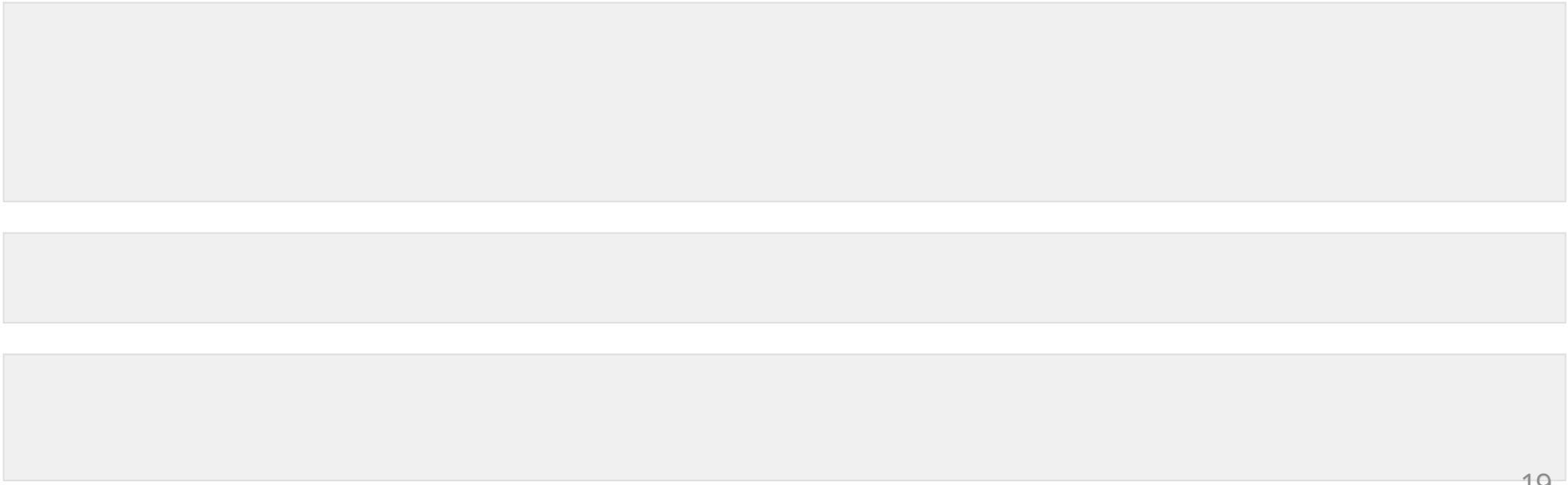
Our boring examples cheated: the main function couldn't see the output from the other goroutine.

It was just printed to the screen, where we pretended we saw a conversation.

Real conversations require communication.

Channels

A channel in Go provides a connection between two goroutines, allowing them to communicate.



Using channels

A channel connects the main and boring goroutines so they can communicate.

Run

20

Synchronization

When the main function executes <–c, it will wait for a value to be sent.

Similarly, when the boring function executes c <– value, it waits for a receiver to be ready.

A sender and receiver must both be ready to play their part in the communication. Otherwise we wait until they are.

Thus channels both communicate and synchronize.

An aside about buffered channels

Note for experts: Go channels can also be created with a buffer.

Buffering removes synchronization.

Buffering makes them more like Erlang's mailboxes.

Buffered channels can be important for some problems but they are more subtle to reason about.

We won't need them today.

The Go approach

Don't communicate by sharing memory, share memory by communicating.

"Patterns"

24

Generator: function that returns a channel

Channels are first-class values, just like strings or integers.

Run

25

Channels as a handle on a service

Our boring function returns a channel that lets us communicate with the boring service it provides.

We can have more instances of the service.

Run

26

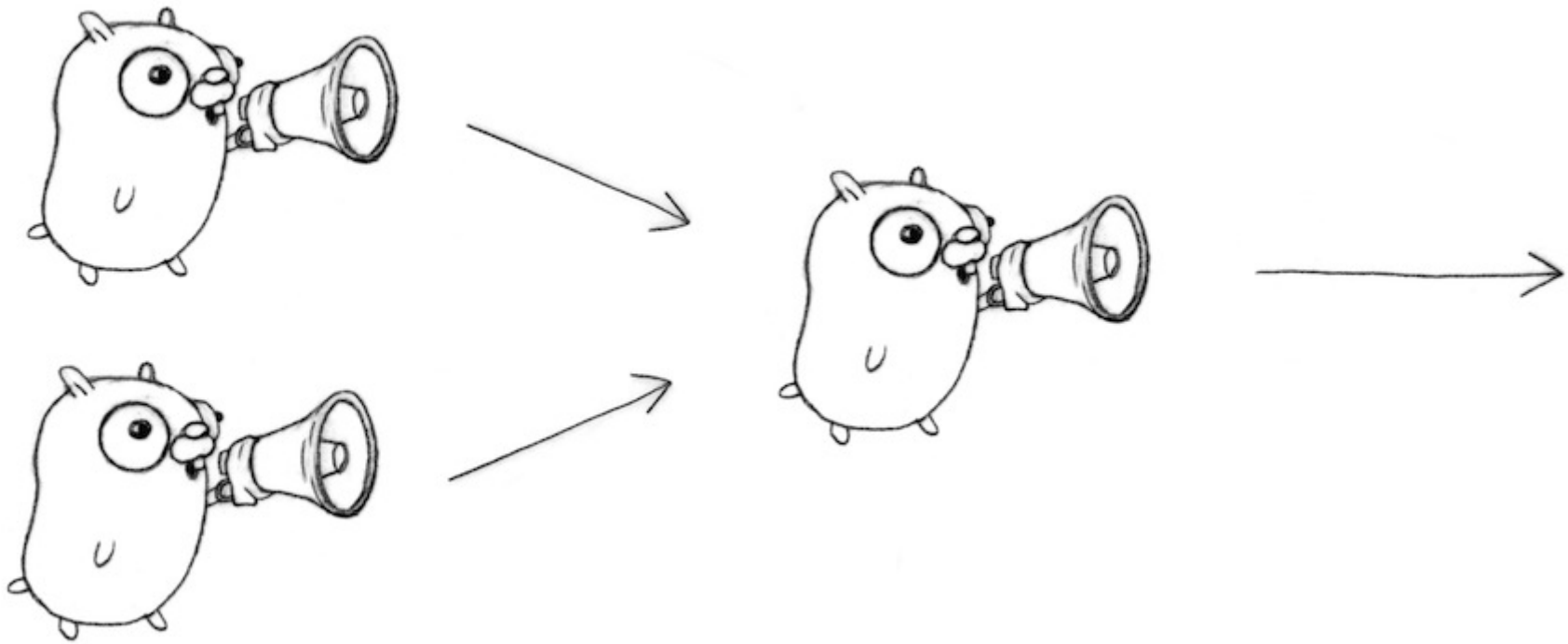
Multiplexing

These programs make Joe and Ann count in lockstep.
We can instead use a fan-in function to let whosoever is ready talk.

Run

27

Fan-in



Restoring sequencing

Send a channel on a channel, making goroutine wait its turn.

Receive all messages, then enable them again by sending on a private channel.

First we define a message type that contains a channel for the reply.

Restoring sequencing.

Each speaker must wait for a go-ahead.

Run

30

Select

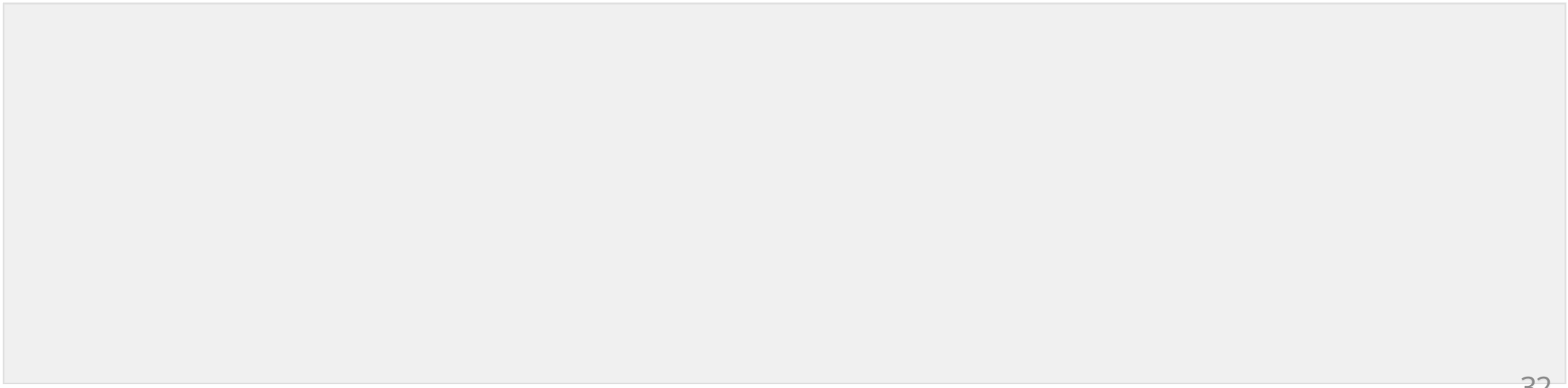
A control structure unique to concurrency.

The reason channels and goroutines are built into the language.

31

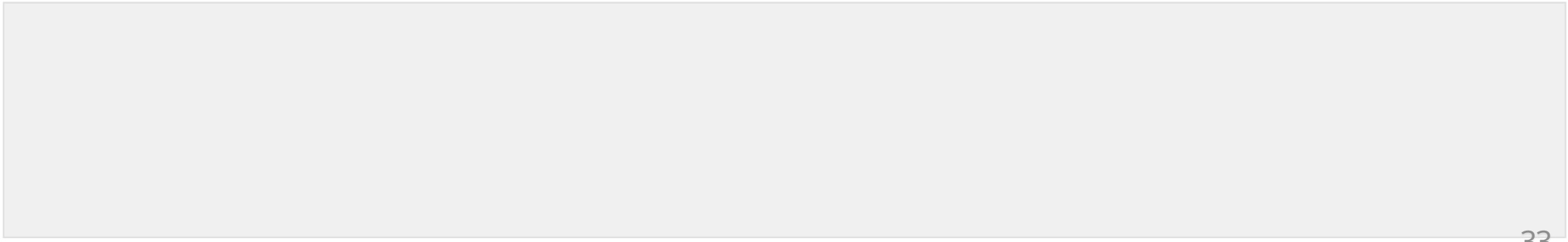
Select

The select statement provides another way to handle multiple channels.
It's like a switch, but each case is a communication:
- All channels are evaluated.
- Selection blocks until one communication can proceed, which then does.
- If multiple can proceed, select chooses pseudo-randomly.
- A default clause, if present, executes immediately if no channel is ready.



Fan-in again

Rewrite our original fanIn function. Only one goroutine is needed. Old:



Fan-in using select

Rewrite our original fanIn function. Only one goroutine is needed. New:

Run

34

Timeout using select

The time.After function returns a channel that blocks for the specified duration. After the interval, the channel delivers the current time, once.

Run

35

Timeout for whole conversation using select

Create the timer once, outside the loop, to time out the entire conversation.
(In the previous program, we had a timeout for each message.)

Run

36

Quit channel

We can turn this around and tell Joe to stop when we're tired of listening to him.

Run

37

Receive on quit channel

How do we know it's finished? Wait for it to tell us it's done: receive on the quit channel

Run

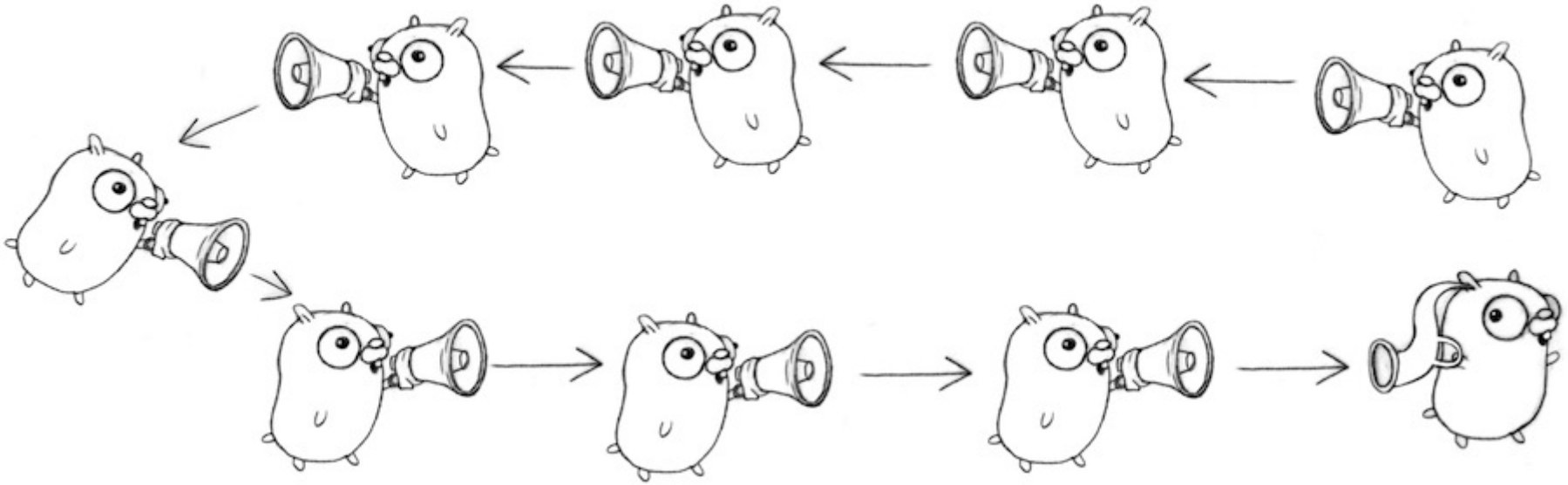
38

Daisy-chain

Run

39

Chinese whispers, gopher style



Systems software

Go was designed for writing systems software.
Let's see how the concurrency features come into play.

41

Example: Google Search

Q: What does Google search do?

A: Given a query, return a page of search results (and some ads).

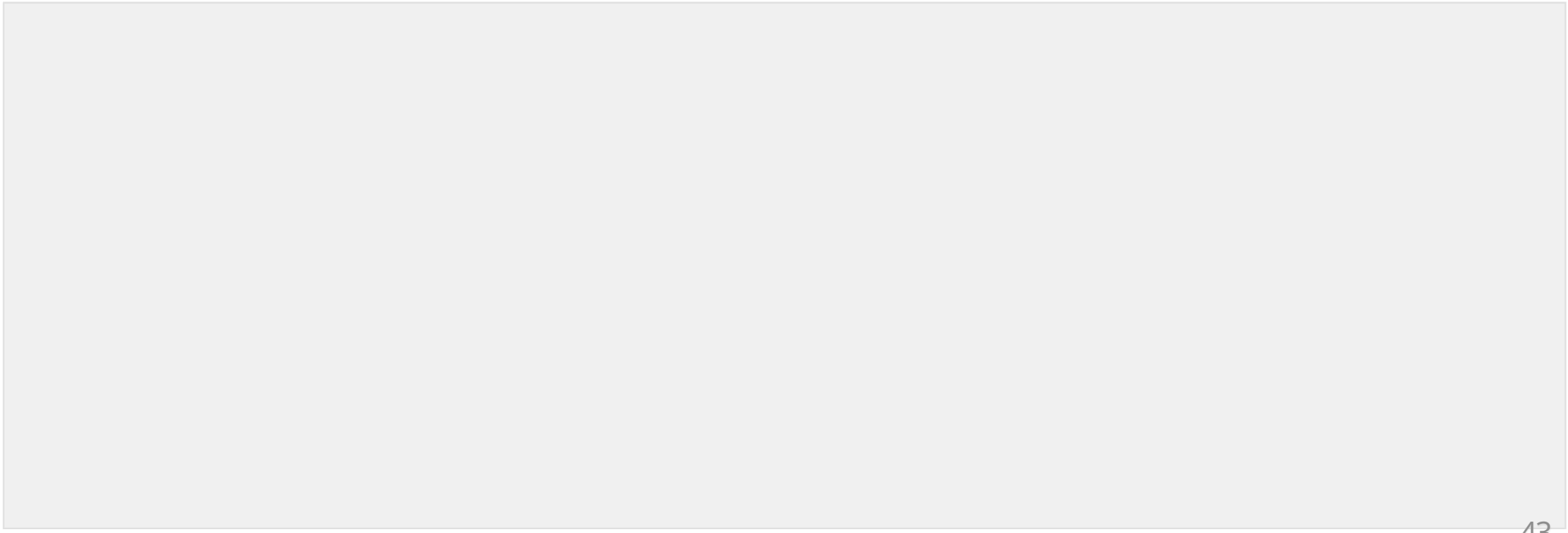
Q: How do we get the search results?

A: Send the query to Web search, Image search, YouTube, Maps, News,etc., then mix the results.

How do we implement this?

Google Search: A fake framework

We can simulate the search function, much as we simulated conversation before.



Google Search: Test the framework

Run

44

Google Search 1.0

The Google function takes a query and returns a slice of Results (which are just strings).

Google invokes Web, Image, and Video searches serially, appending them to the results slice.

Run

45

Google Search 2.0

Run the Web, Image, and Video searches concurrently, and wait for all results.

No locks. No condition variables. No callbacks.

Run

46

Google Search 2.1

Don't wait for slow servers. No locks. No condition variables. No callbacks.

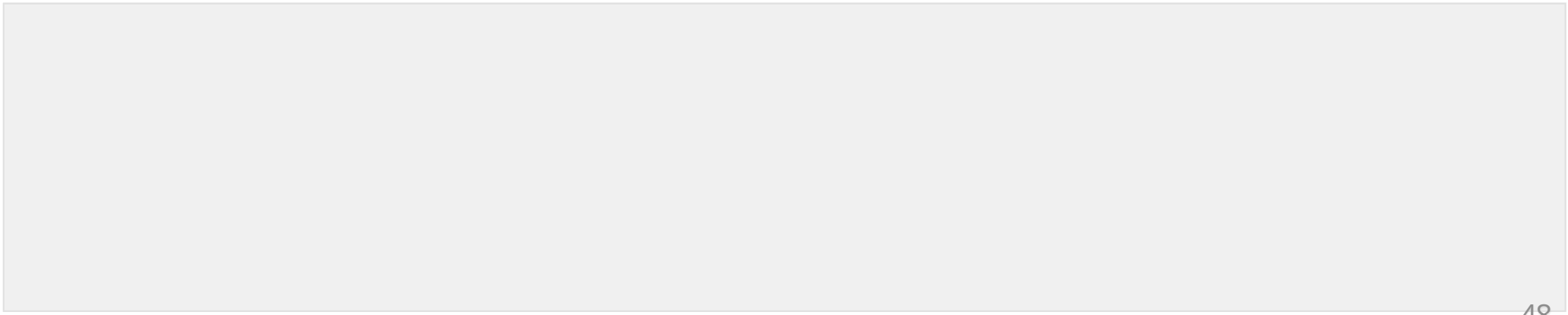
Run

47

Avoid timeout

Q: How do we avoid discarding results from slow servers?

A: Replicate the servers. Send requests to multiple replicas, and use the first response.



Using the First function

Run

49

Google Search 3.0

Reduce tail latency using replicated search servers.

Run

50

And still...

No locks. No condition variables. No callbacks.

51

Summary

In just a few simple transformations we used Go's concurrency primitives to convert a

- slow
- sequential
- failure-sensitive

program into one that is

- fast
- concurrent
- replicated
- robust.

More party tricks

There are endless ways to use these tools, many presented elsewhere.

Chatroulette toy:

golang.org/s/chat-roulette (http://golang.org/s/chat-roulette)

Load balancer:

golang.org/s/load-balancer (http://golang.org/s/load-balancer)

Concurrent prime sieve:

golang.org/s/prime-sieve (http://golang.org/s/prime-sieve)

Concurrent power series (by McIlroy):

golang.org/s/power-series (http://golang.org/s/power-series)

Don't overdo it

They're fun to play with, but don't overuse these ideas.

Goroutines and channels are big ideas. They're tools for program construction.

But sometimes all you need is a reference counter.

Go has "sync" and "sync/atomic" packages that provide mutexes, condition variables, etc. They provide tools for smaller problems.

Often, these things will work together to solve a bigger problem.

Always use the right tool for the job.

Conclusions

Goroutines and channels make it easy to express complex operations dealing with

- multiple inputs
- multiple outputs
- timeouts
- failure

And they're fun to use.

Links

Go Home Page:

golang.org (http://golang.org)

Go Tour (learn Go in your browser)

tour.golang.org (http://tour.golang.org)

Package documentation:

golang.org/pkg (http://golang.org/pkg)

Articles galore:

golang.org/doc (http://golang.org/doc)

Concurrency is not parallelism:

golang.org/s/concurrency-is-not-parallelism (http://golang.org/s/concurrency-is-not-parallelism)

Thank you

Rob Pike
Google
http://golang.org/s/plusrob (http://golang.org/s/plusrob)
@rob_pike (http://twitter.com/rob_pike)
http://golang.org (http://golang.org)