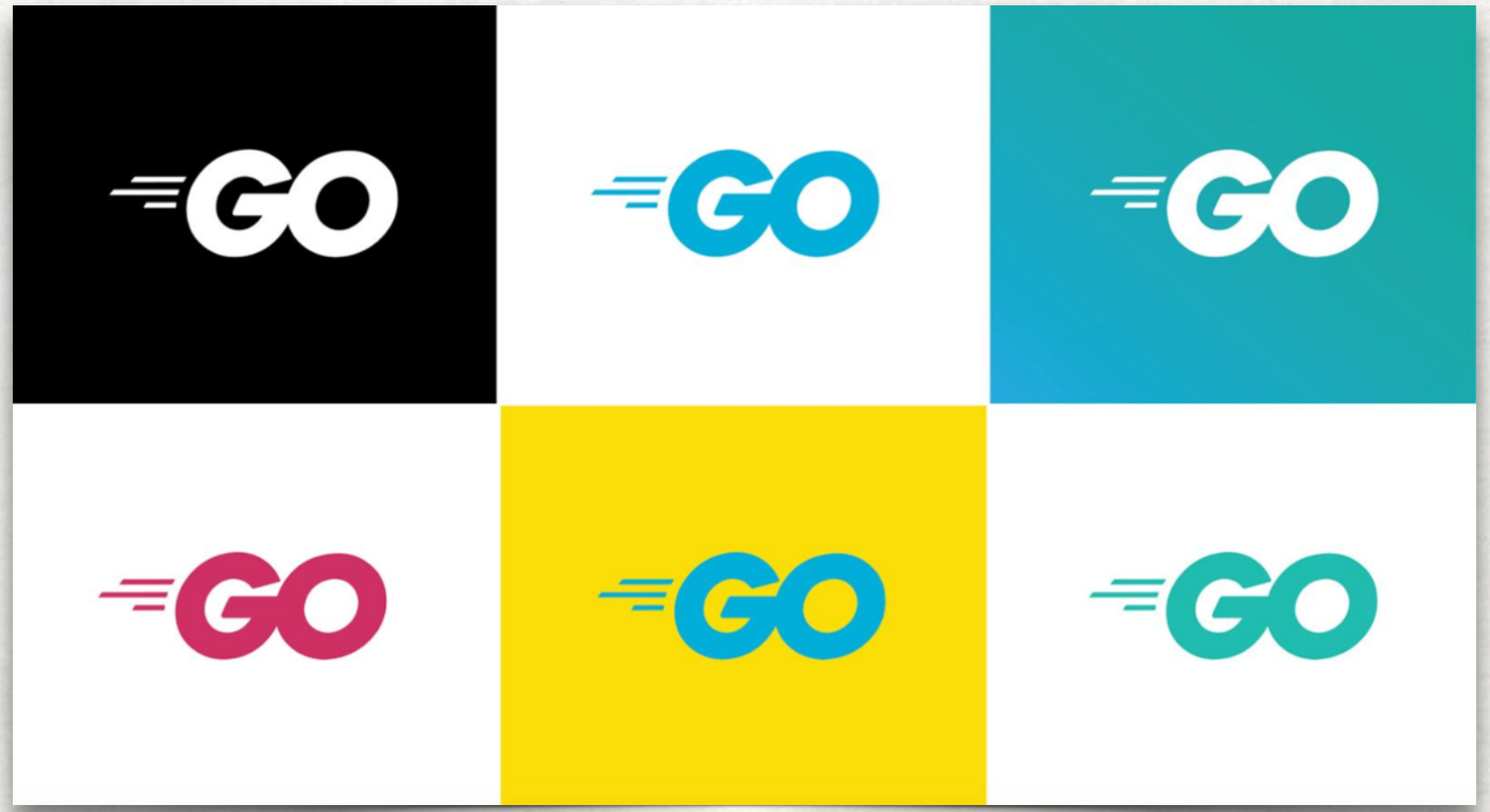


CONCURRENCY IN GO: PATTERNS AND VISUALIZATIONS

DOMINIC GAGNÉ

JANUARY 29, 2019

TODAY'S TALK

- Concurrency Basics
- Go's Approach to Concurrency
- Visualizing Concurrency
- Concurrency Patterns

WHAT IS CONCURRENCY?

1 2 next

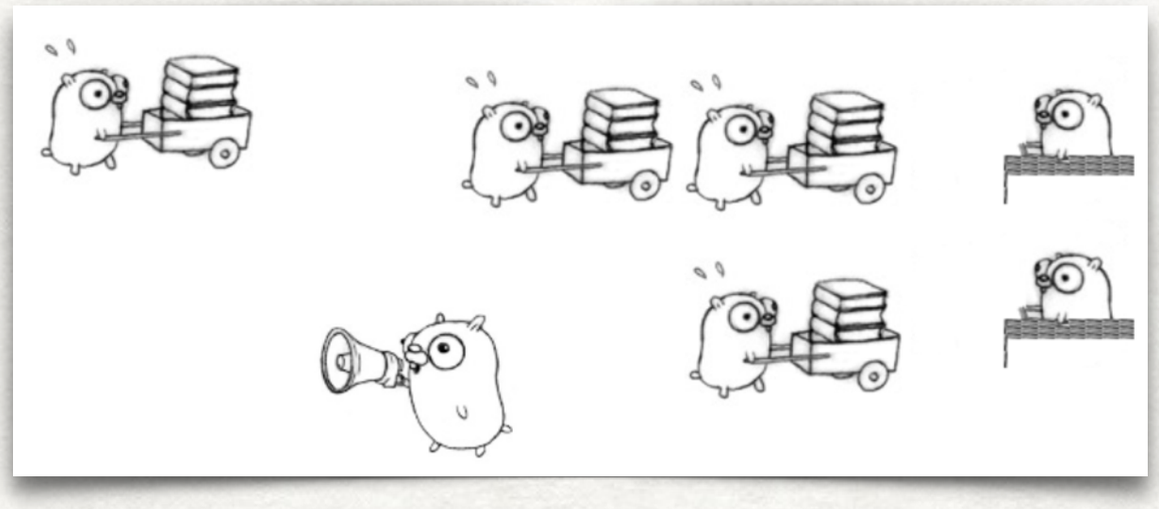
1020

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, *multitasking* on a single-core machine.

Parallelism is when tasks *literally* run at the same time, e.g., on a multicore processor.

Quoting Sun's *Multithreaded Programming Guide*:

CONCURRENCY BASICS



- Concurrent programs are about a division of labour
- Concurrency involves managing separate tasks at runtime
- Properly written concurrent programs can take advantage of multi-core machines
- https://blog.golang.org/concurrency-is-not-parallelism

GO'S APPROACH TO CONCURRENCY

- Concurrency primitives are part of the language, no libraries or dependencies needed
- Concurrency is achieved via *goroutines* (like lightweight threads)
- Communication and synchronization achieved via *channels*
- Ideal for distributed systems

IN A NUTSHELL

"Don't communicate by sharing memory,
share memory by communicating."
- Rob Pike

TOOLS FOR VISUALIZING CONCURRENCY

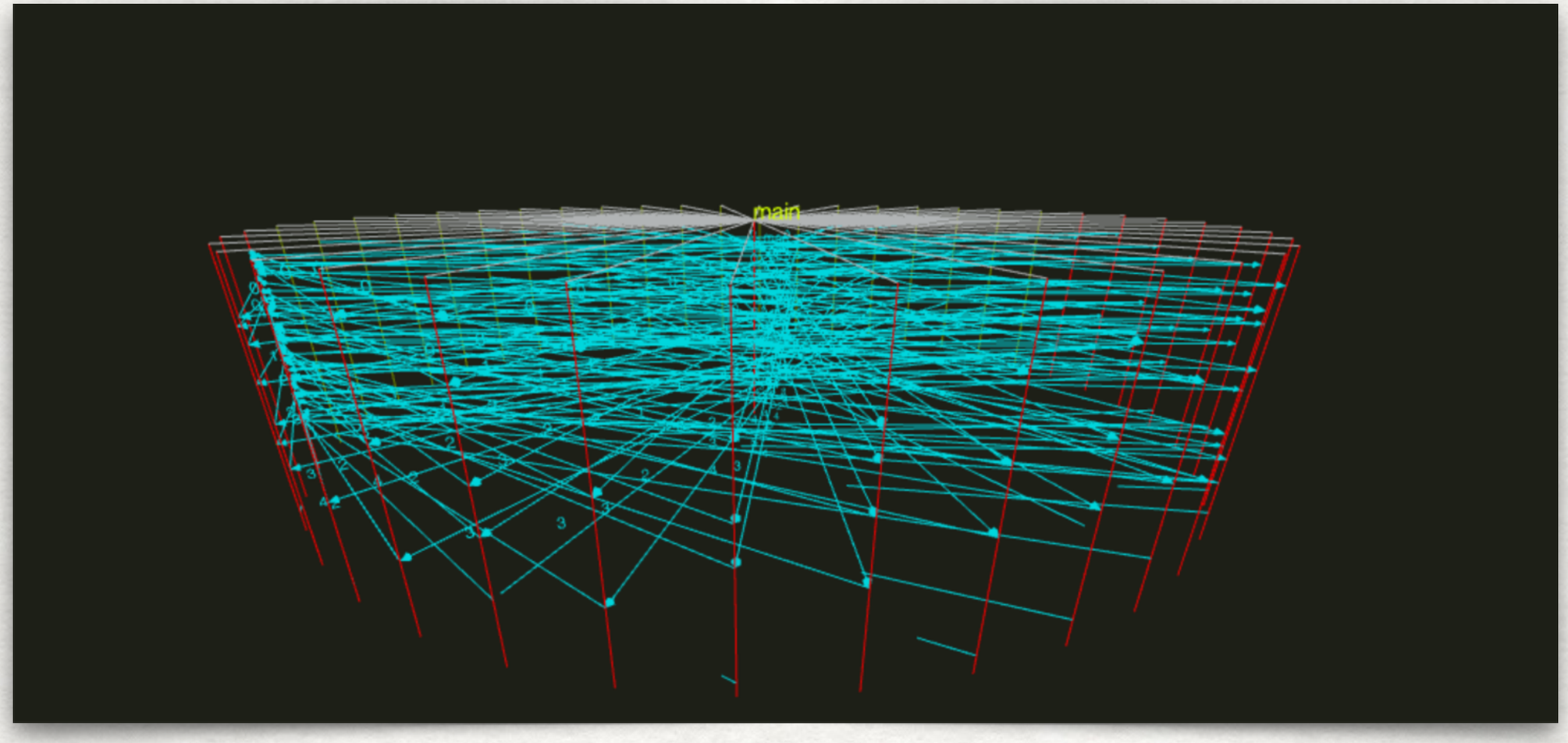
- Ivan's tool uses the output from a system trace as input to WebGL to produce animations of goroutines communicating

- Very useful. But hard to use, unreliable, does not work with recent versions of Go, and is now unmaintained

Ivan Danyliuk,
creator of the original
Go visualization tool

Tool to make Ivan's tool usable again:
https://github.com/DominicGagne/Go-Concurrency-Visualizer

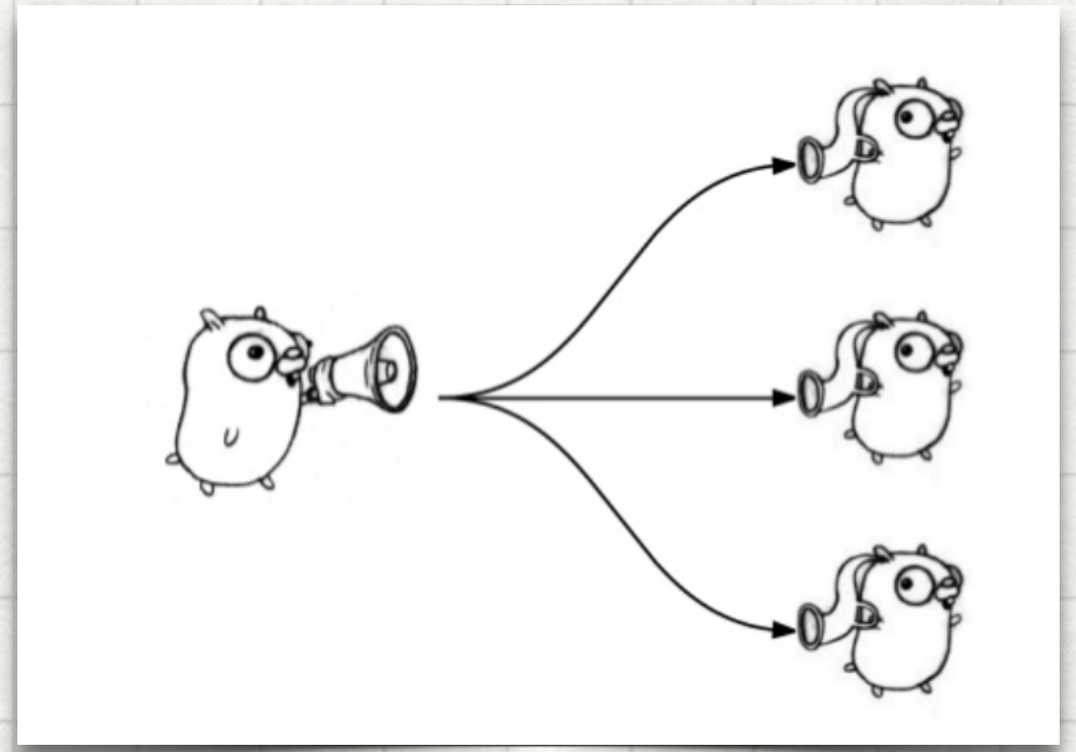
VISUALIZING CONCURRENCY



- Green: unblocked, working
- Red: blocked, waiting on another goroutine
- Grey: Spawning or terminating goroutine
- Blue: Value being passed along a channel, from one goroutine to another

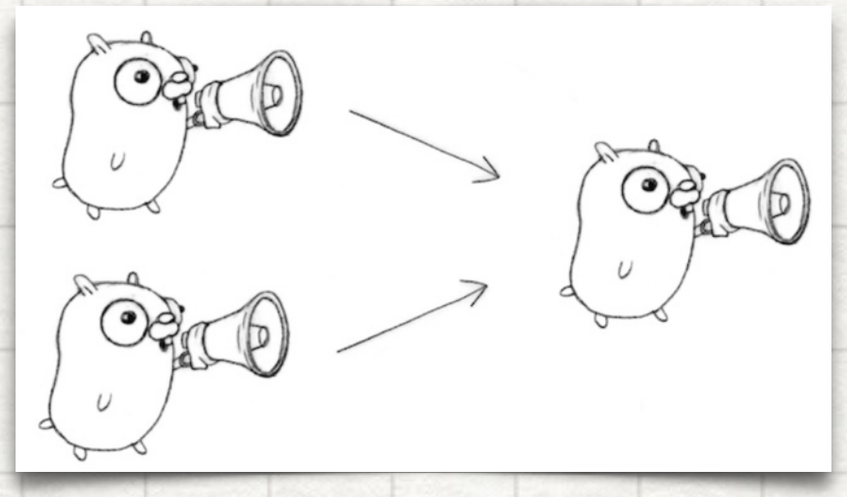
FAN-OUT

- Fan-out is a pattern used to distribute work to multiple goroutines
- This pattern is used to make efficient use of multiple cores for CPU bound tasks, and to enable many I/O bound tasks to occur simultaneously



FAN-IN

- Fan-in is used to gather data from multiple goroutines and merge that data onto a single channel
- This pattern is especially useful for collecting and organizing data that has been spread to a large number of goroutines via fan-out



```
// receive a variadic number of channels, merge all their values
// onto a single channel, and pass that channel back as read-only
func merge(channels ...chan int) <-chan int {
    // aggregate is where all output of these channels
    // gets funneled into
    outBound := make(chan int)

    wg := &sync.WaitGroup{}
    wg.Add(len(channels))

    output := func(dataSource chan int) {
        for val := range dataSource {
            outBound <- val
        }
```

PIPELINES

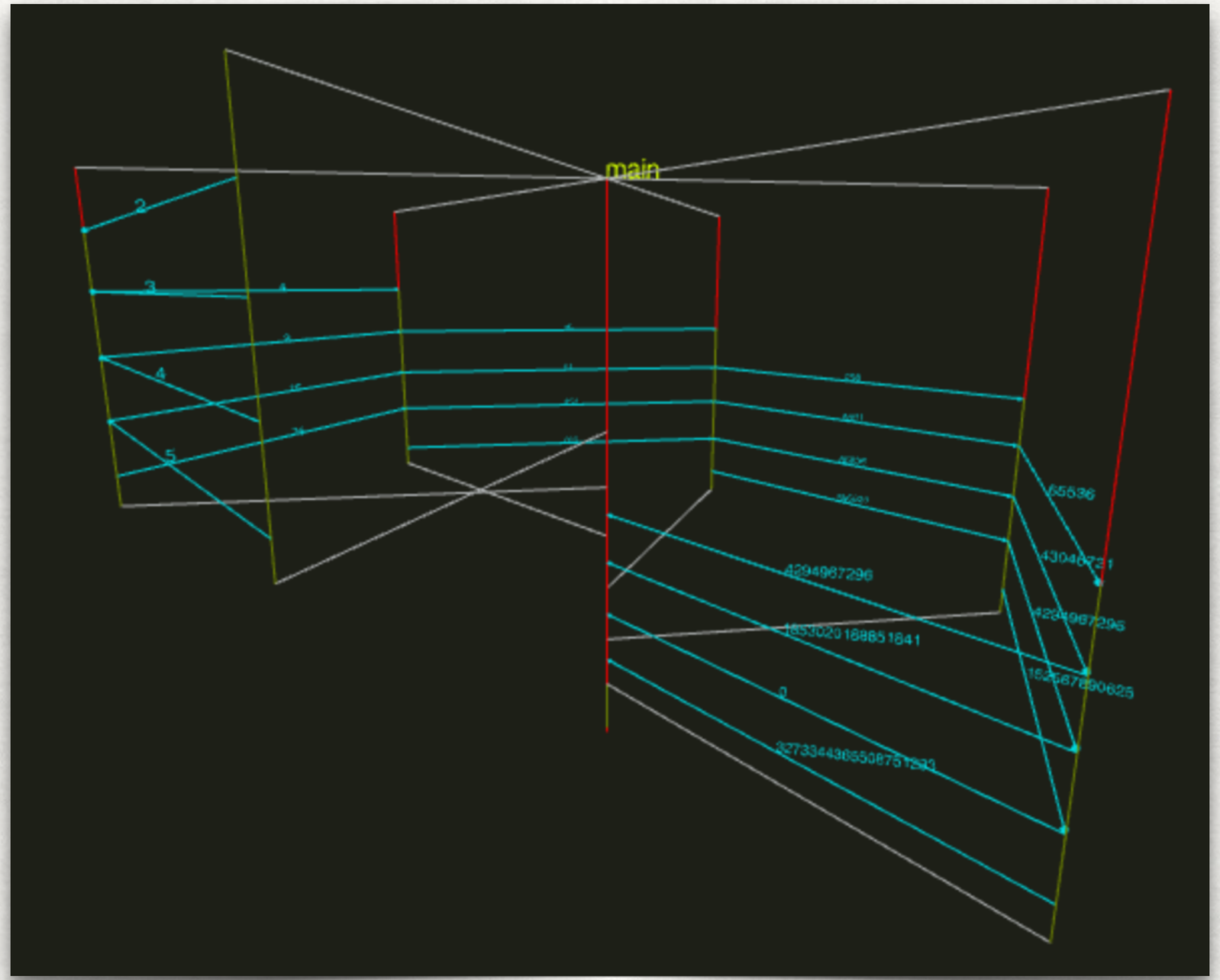
- Pipelines are used to separate pieces of an algorithm into smaller chunks, so that they may occur concurrently
- Data flows from one stage to the next
- Useful for processing large streams of data without using much memory

```go
// copied from https://blog.golang.org/pipelines
// converts variadic or slice of ints to distinct values in a pipeline
func gen(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        for _, n := range nums {
            time.Sleep(time.Millisecond * 100)
            out <- n
        }
        // small sleep before terminating the goroutine to ensure
        // trace output is collected
        time.Sleep(time.Millisecond * 50)
        // inform the pipeline that's all we've got
        close(out)
    }()

    return out
}
```

Single stage of a pipeline,
taken from http://blog.golang.org/pipelines

VISUALIZING PIPELINES



Data flowing from one stage of the pipeline to the next via channels

THE SUBSCRIBER PATTERN

- The subscriber pattern is a method used to ensure the integrity of a list of goroutines (listening on channels) subscribed to a particular event.

- At any given time, the list of subscribers can either be amended, or the subscribers can be notified of an event, but not both

- Made possible because of the *select statement*

THE SELECT STATEMENT

- The single most powerful feature in Go for controlling concurrently executing operations
- Only a single case can be executing at any given time
- “Switch statement, but for communication" - Rob Pike

SLOW CONSUMERS

- Slow or dead consumers can decrease the performance of the subscriber pattern
- It's important for the main goroutine in a subscriber pattern to defend itself

```go
func (n *notifier) run() {
    for {
        select {
        case event := <-n.events:
            fmt.Printf("received a new event: %+v\n", event)

            // inform subscribers
            for id, sub := range n.subscribers {
                select {
                case sub <- event:
                    // subscriber has received the message, carry on to the next one
                case <-time.After(time.Millisecond * 500):
                    // subscriber has not responded in 500ms, kill that consumer
                    delete(n.subscribers, id)
                }
            }
        case sub := <-n.subscribeInternal:
            n.subscribers[len(n.subscribers)] = sub
        }
    }
```

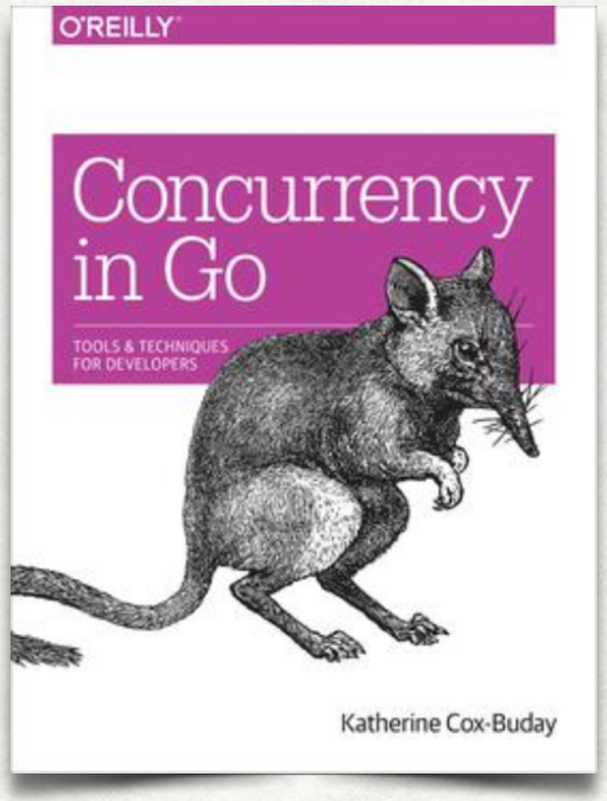
HELPFUL RESOURCES

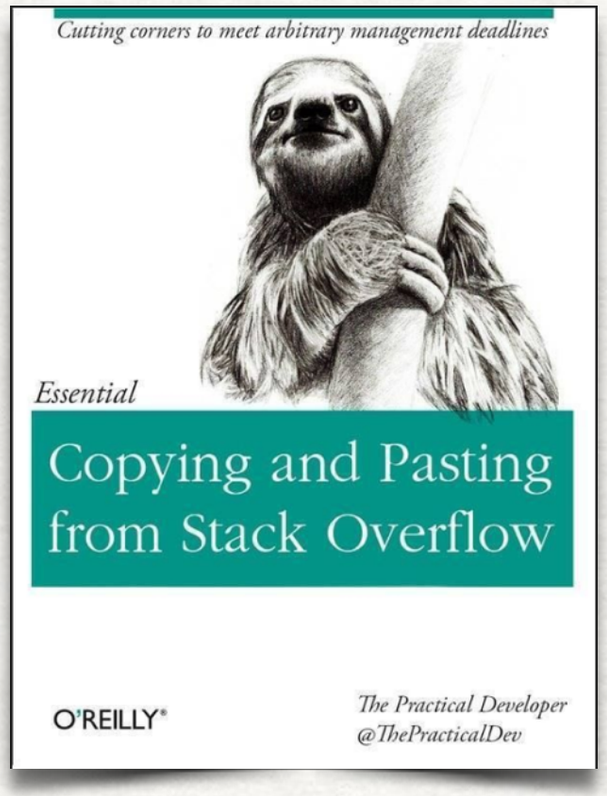
Rob Pike - 'Concurrency Is Not Parallelism'

https://www.youtube.com/watch?v=cN_DpYBzKso

GopherCon 2016: Ivan Danyliuk - Visualizing Concurrency in Go

https://www.youtube.com/watch?v=KyuFeiG3Y60





Concurrency Visualizer Source Code

https://github.com/DominicGagne/Go-Concurrency-Visualizer