Concurrency Patterns and Mutexes

DD1396
Ric Glassey
glassey@kth.se

Concepts

Concurrency Patterns

Explicit Locking with Mutexes

Concurrency Patterns

Patterns

Patterns are **reusable solutions**

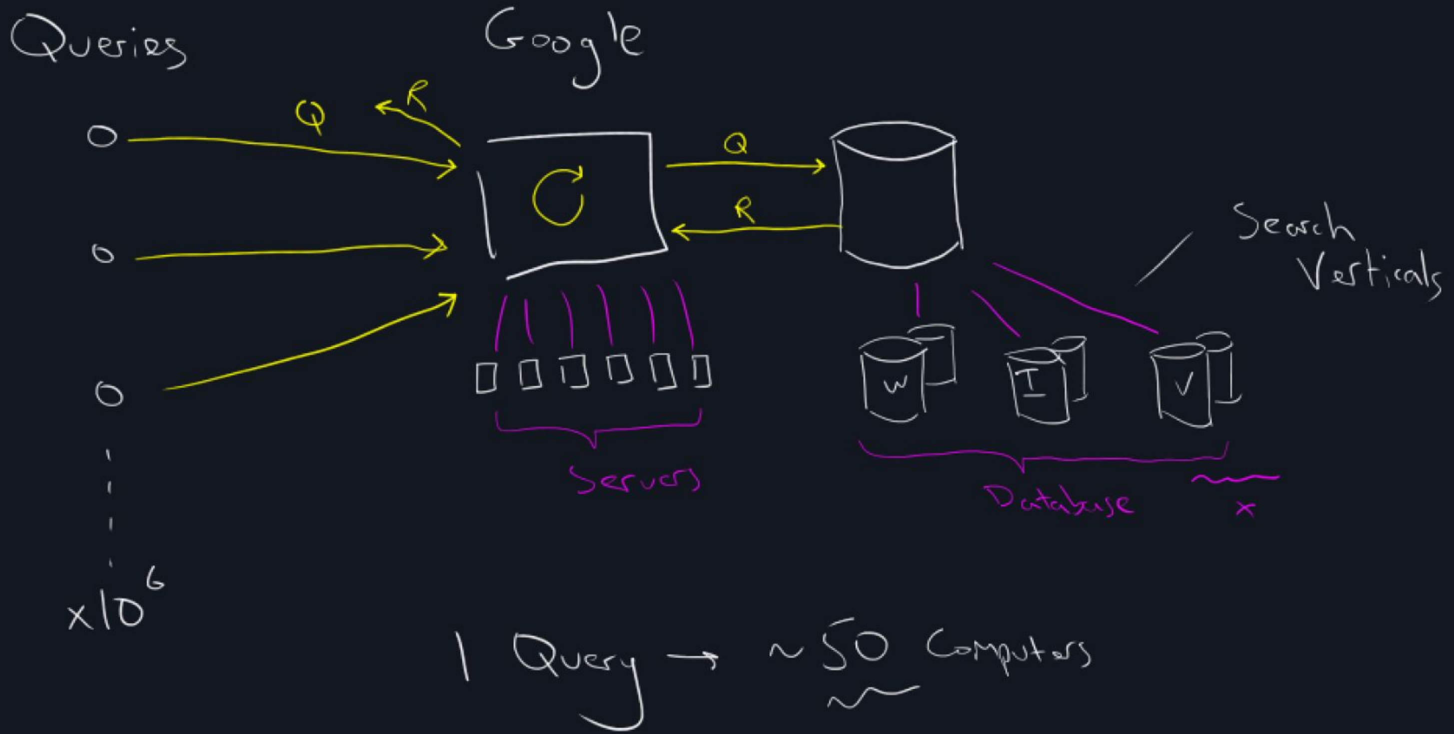
- Design patterns
- Architectural patterns
- Antipatterns

Concurrency patterns

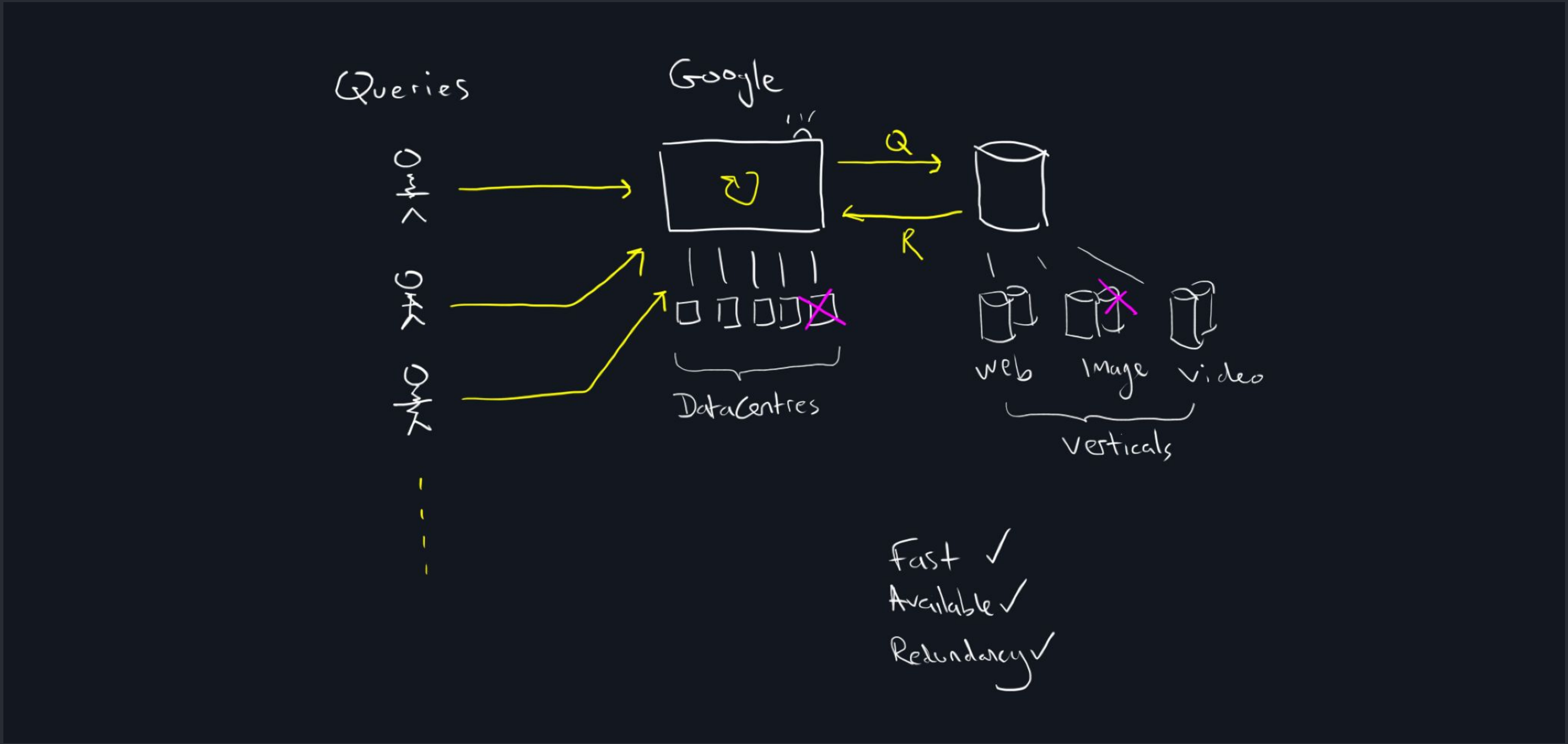
- **Fan-in**
- **Timeout**
- **Replication**

Google Search in Go

Draw: Abstract Google



Queries
Google
Q
R
Q
R
Search Verticals
W
I
V
Servers
Database
x
$\times 10^6$
1 Query → ~50 Computers



Queries
Google
Q
R
DataCentres
web
Image
video
Verticals
Fast ✓
Available ✓
Redundancy ✓

[Video]

Search verticals

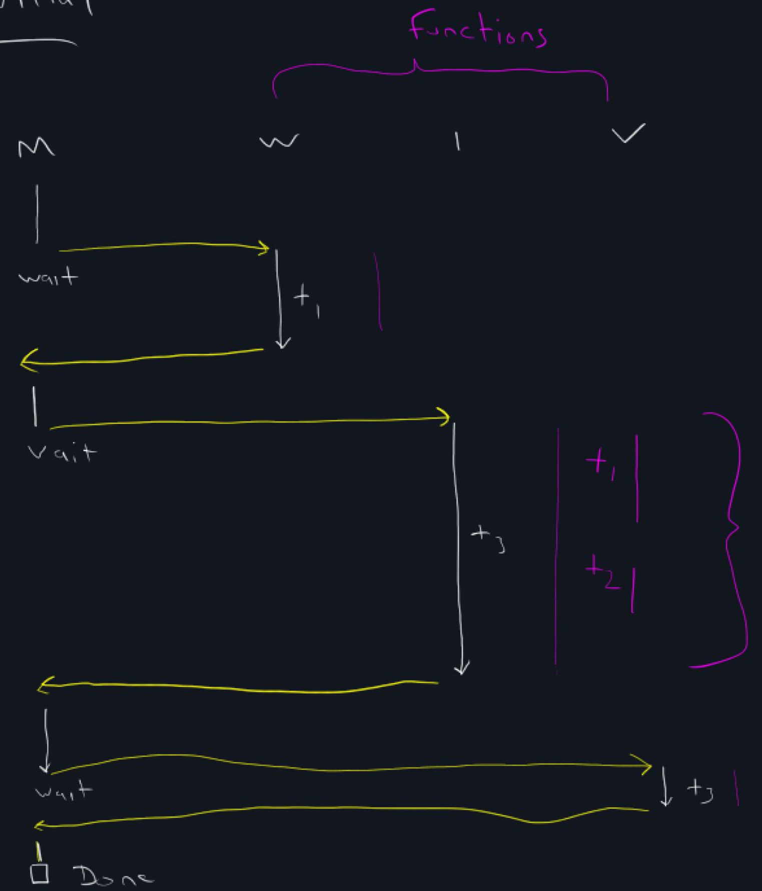
```go
type Result string

type Search func(query string) Result

var (
    // search “verticals” modelled as functions
    Web = fakeSearch("web")
    Image = fakeSearch("image")
    Video = fakeSearch("video")
)

func fakeSearch(kind string) Search {
    // factory function to create a search vertical function
    return func(query string) Result {
        // fake search that just waits a few milliseconds
        time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
        return Result(fmt.Sprintf("%s result for %q\n", kind, query))
    }
}
```

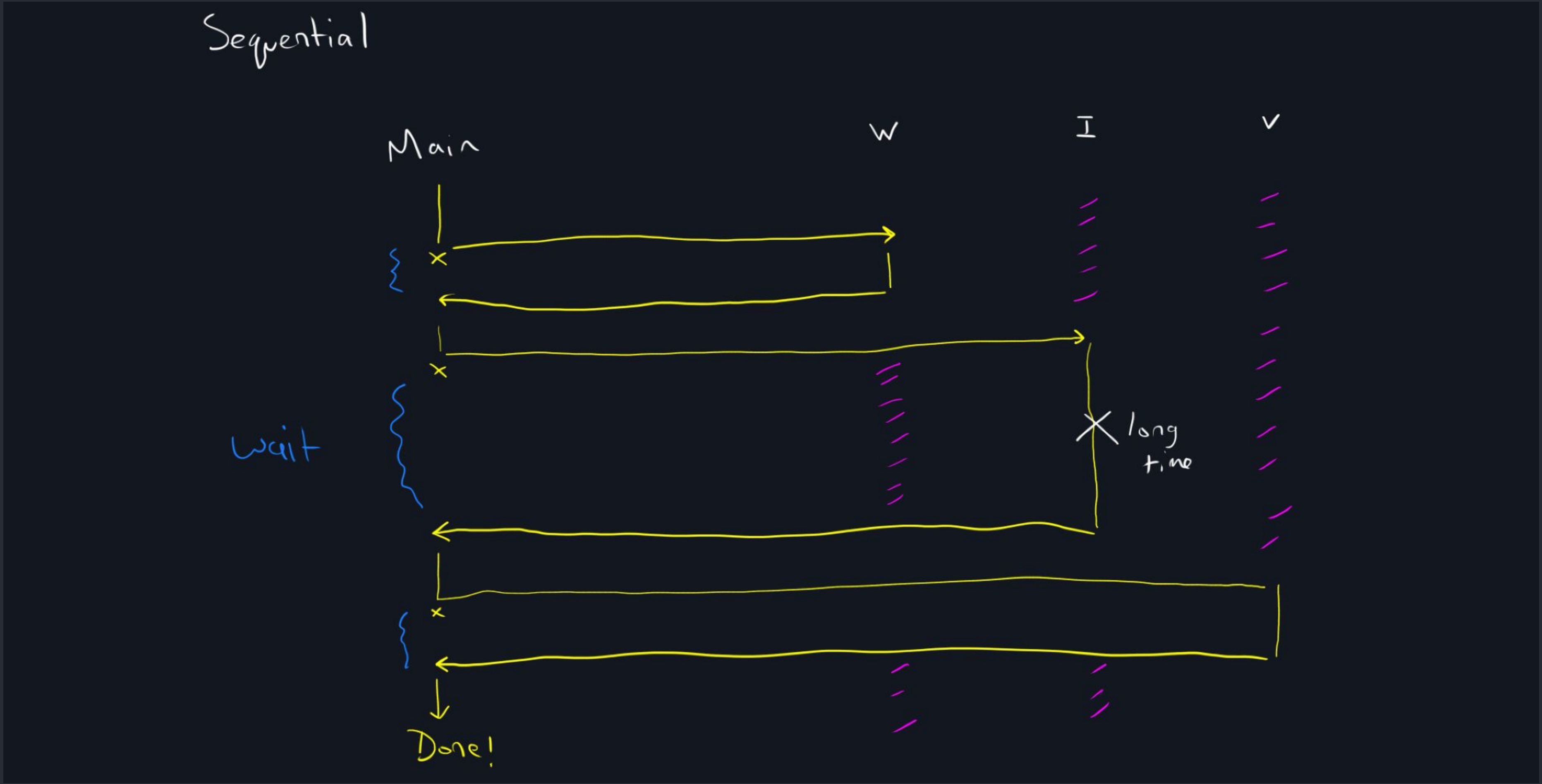
Execute a search

```
func Google(query string) (results []Result) {
    // Search for Web, Image and Video results
    results = append(results, Web(query))
    results = append(results, Image(query))
    results = append(results, Video(query))
    return
}

func main() {
    // Time the search
    rand.Seed(time.Now().UnixNano())
    start := time.Now()
    results := Google("golang")
    elapsed := time.Since(start)
    fmt.Println(results)
    fmt.Println(elapsed)
}
```

Draw: Sequential Search

Sequential





[Video]

Demo: Sequential Search

Sequential slowdown

Each search vertical is called in a **linear sequence**

- May have different **performance**
- May have different **running conditions**

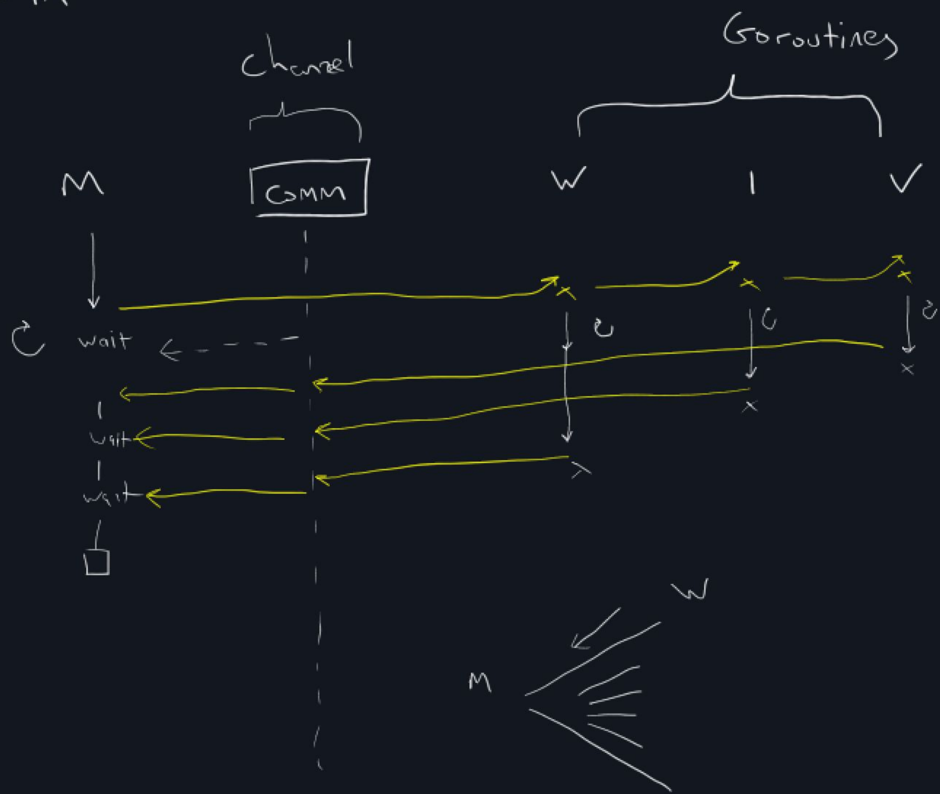
Opportunity for **concurrency**

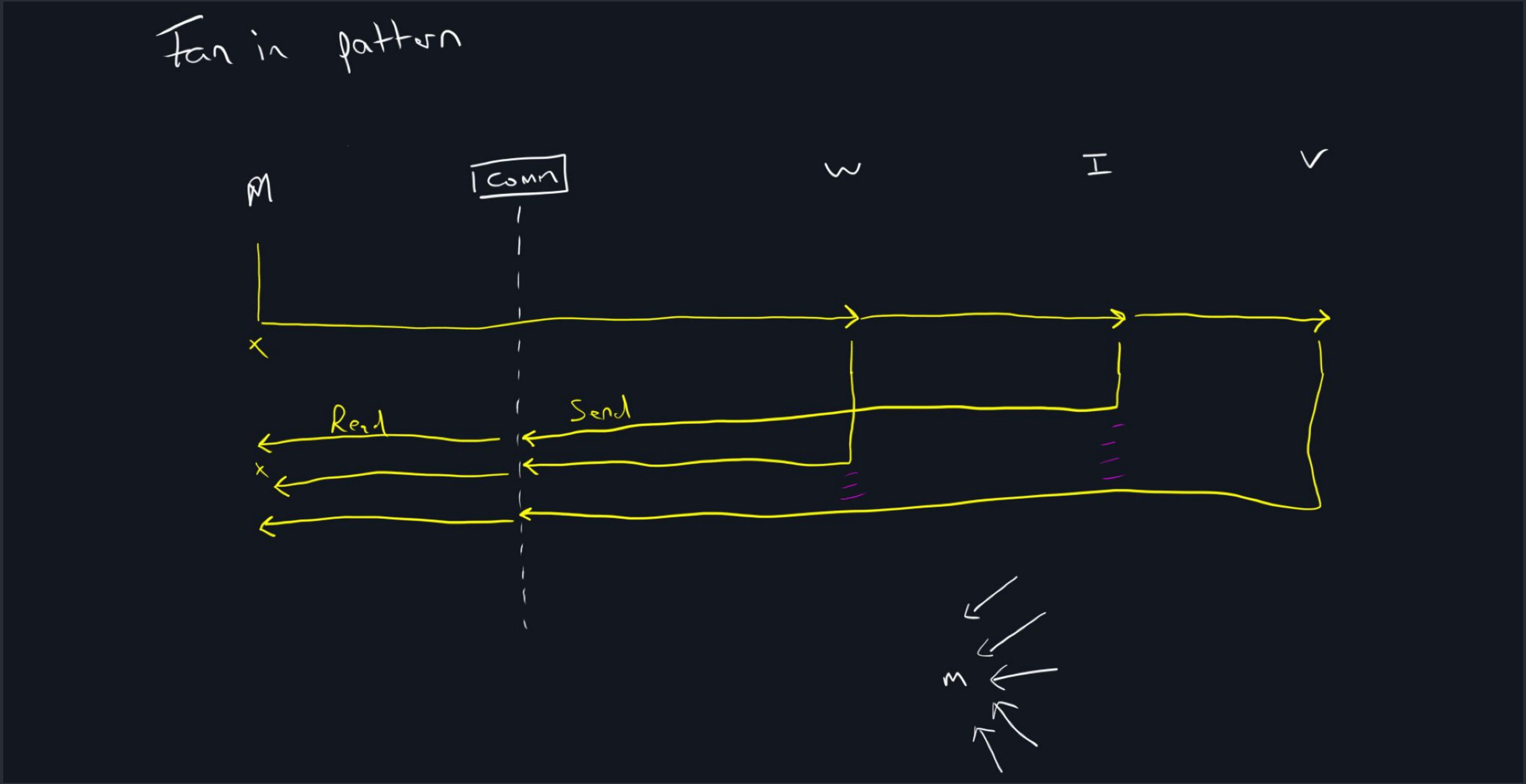
- Search verticals are waiting to be called
- Runtime will be the sum of all three systems

How can this be improved (using Go)?

Fan-in Pattern

Fan - In





Video

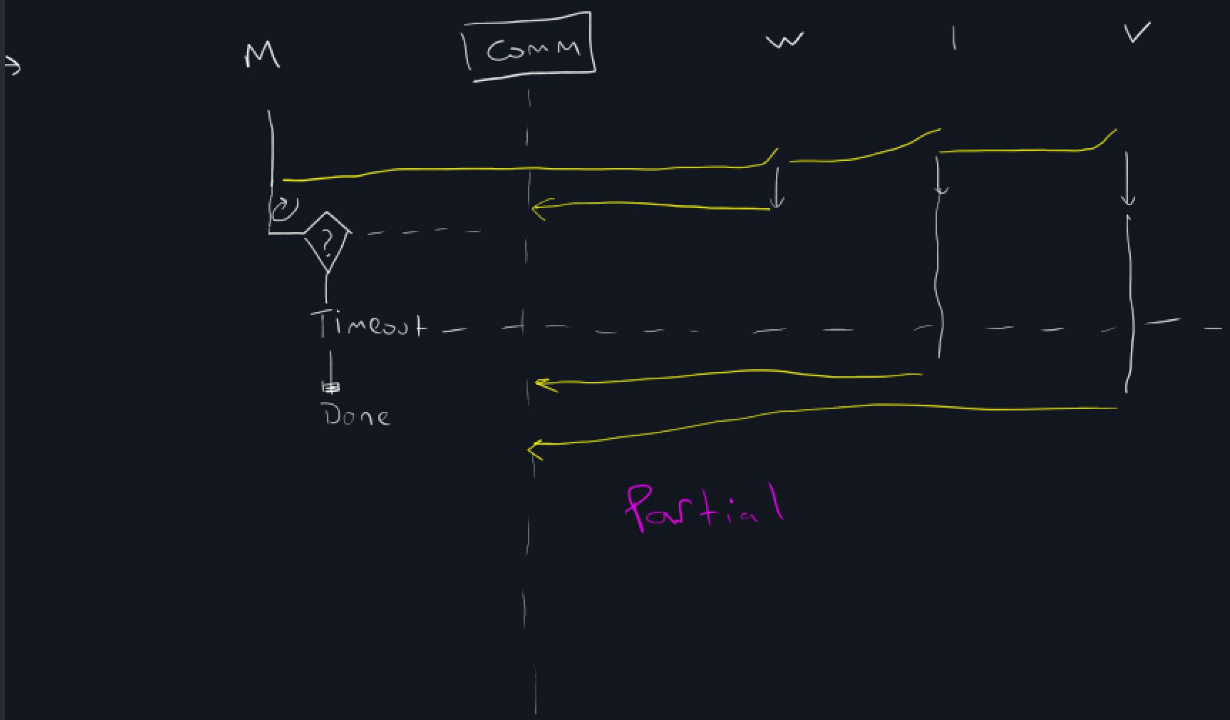
Fan-in

```go
func Google(query string) (results []Result) {
    // Use a channel to collect results
    comm := make(chan Result)

    // create three search threads using a fan-in pattern
    go func() { comm <- Web(query) } ()
    go func() { comm <- Image(query) } ()
    go func() { comm <- Video(query) } ()

    // collect results
    for i := 0; i < 3; i++ {
        result := <- comm
        results = append(results, result)
    }
    return
}
```

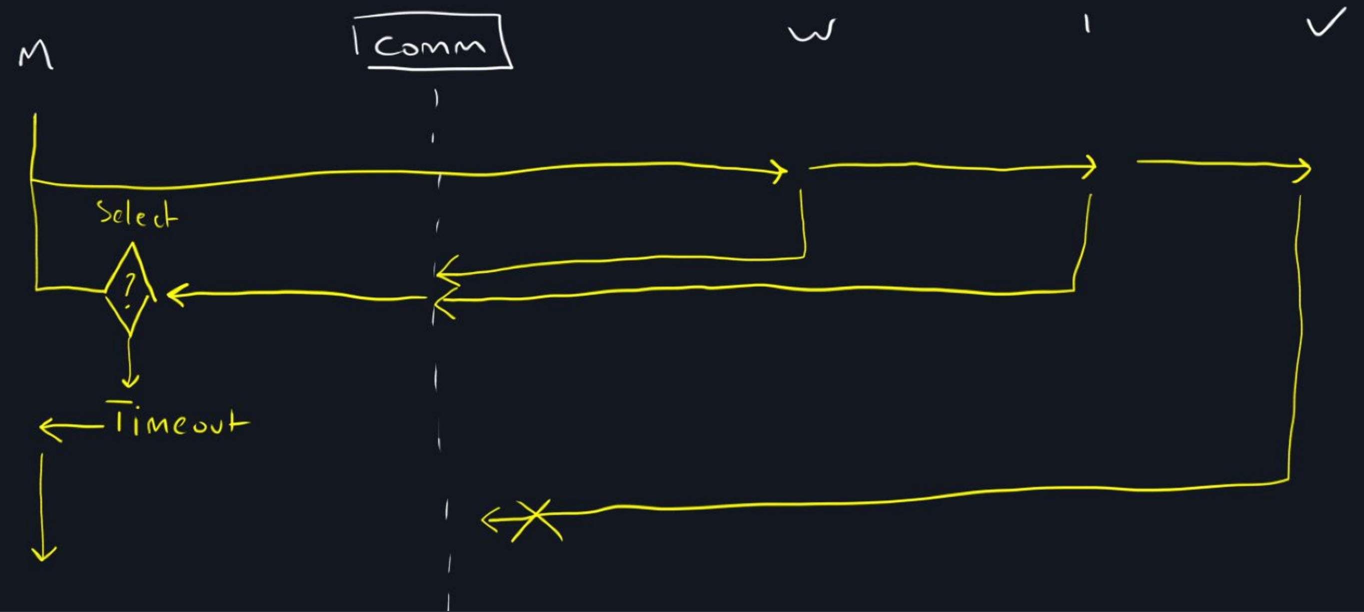
Demo: Fan-in

Time-out Pattern

Time-out



Timeout



Video

Time-out

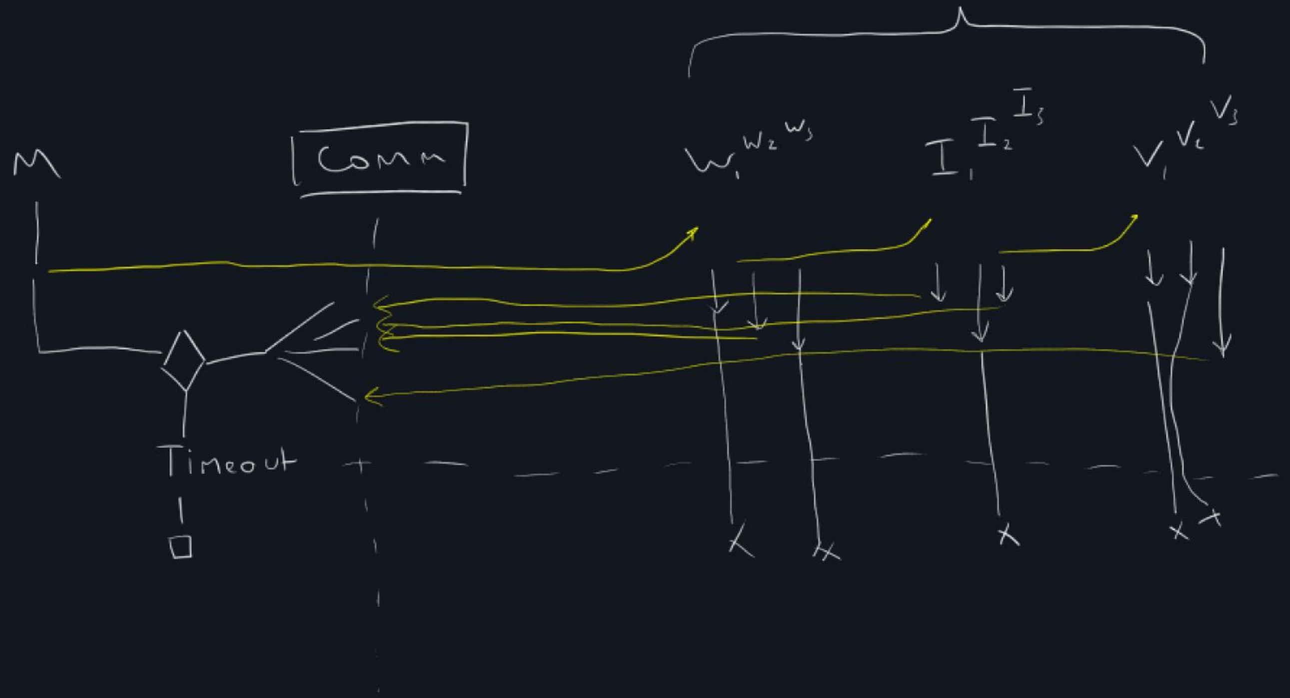
```go
func Google(query string) (results []Result) {
    comm := make(chan Result)

    // create three search threads using a fan-in pattern
    go func() { comm <- Web(query) } ()
    go func() { comm <- Image(query) } ()
    go func() { comm <- Video(query) } ()

    // collect results; but do not wait on slow services
    timeout := time.After(70 * time.Millisecond)
    for i := 0; i < 3; i++ {
        select {
        case result := <- comm:
            results = append(results, result)
        case <- timeout:
            fmt.Println("timed out")
            return
        }
    }
    return
}
```

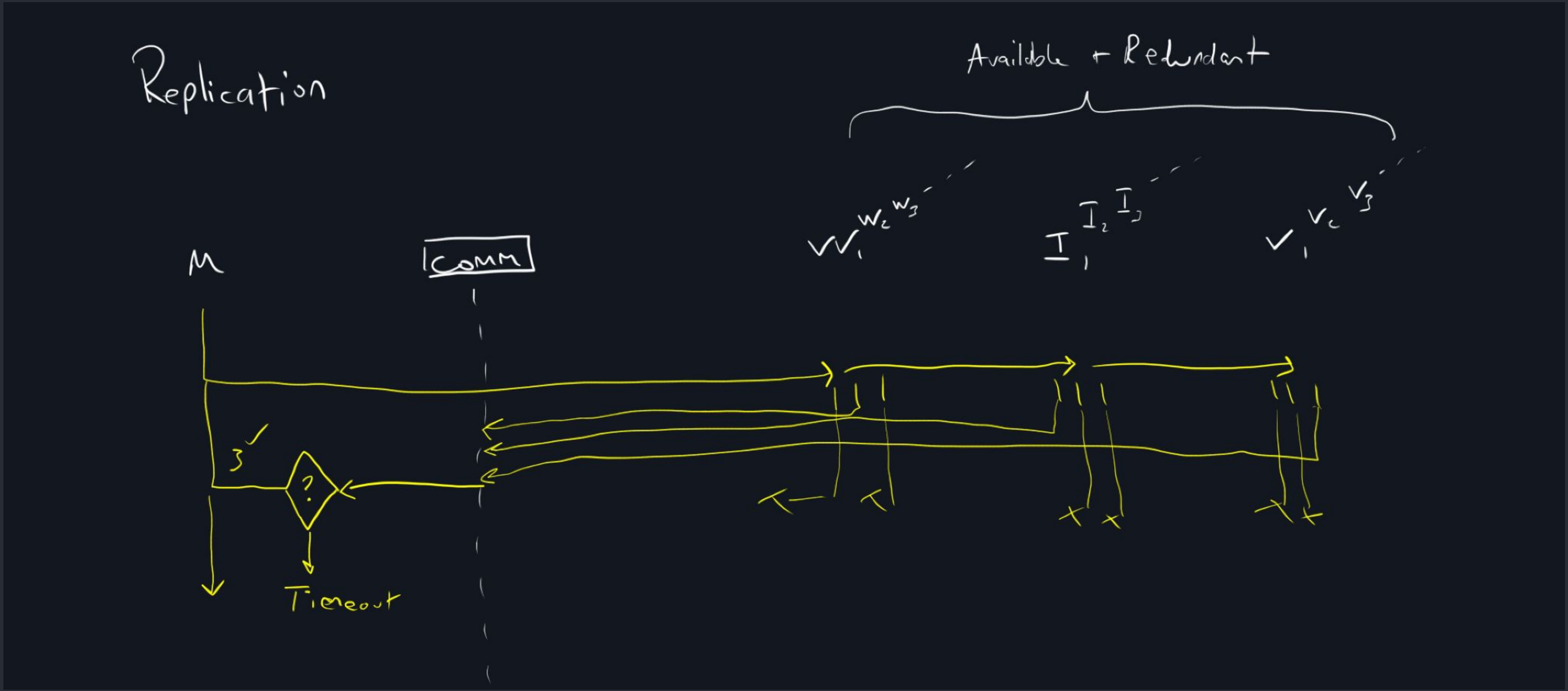
Demo: Time-out Pattern

Replication Pattern

Replication

Redundant, Replicated, Inexpensive





Video

Replication

```go
func Google(query string) (results []Result) {
    comm := make(chan Result)

    // create three search threads using a fan-in pattern
    go func() { comm <- First(query, Web1, Web2) } ()
    go func() { comm <- First(query, Image1, Image2) } ()
    go func() { comm <- First(query, Video1, Video2) } ()
    return
}

func First(query string, replicas ...Search) Result {
    // launch replicas and return fastest response
    c := make(chan Result)
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

Demo: Replication

Complex system; Low cost

Modern scalable systems need to be

- Fast
- Available
- Redundant

Go provides the concurrency primitives

Concurrency pattern provides a solution

Locking by Mutual Exclusion

Simple is better

Race conditions are a problem of concurrent systems

Go can share variables over channels as one solution

The classic solution is **mutual exclusion**

- Lock a variable
- Only one thread can have write access
- "No two threads are in their **critical section** at the same time"
- i.e. reading, updating and writing a shared variable

Using a mutex

Mutex is an element within a program that can **control access to shared data** by **locking** and **unlocking**

During critical section (e.g. update):

- Lock is acquired by thread
- Work is done, then
- Lock released for other waiting threads

My First Bank App

```go
type account struct {
    amount    float64
}

func (acc *account) Deposit(sum float64) {
    acc.amount += sum
}

func (acc *account) Withdraw(sum float64) {
    acc.amount -= sum
}

func (acc *account) Balance() string {
    return strconv.FormatFloat(acc.amount, 'f', 2, 64) + " Kr"
}
```

Simulation setup

```go
func main() {

    var joint_account account
    joint_account.Deposit(1000.00)

    // stop main from quitting before threads
    wg := new(sync.WaitGroup)
    wg.Add(2)

    // Concurrent access of joint bank account (what could possibly go wrong?)
    go func () { // Person One
        joint_account.Deposit(50.00)
        joint_account.Deposit(50.00)
        joint_account.Withdraw(200.00)
        joint_account.Deposit(50.00)
        wg.Done()
    }()

    go func () { // Person Two
        joint_account.Deposit(50.00)
        joint_account.Deposit(50.00)
        joint_account.Withdraw(200.00)
        joint_account.Deposit(50.00)
        wg.Done()
    }()

    // ensure both customers have finished
    wg.Wait()
    fmt.Println(joint_account.Balance())
}
```

Demo: My First Bank App

Detecting Race Conditions

Runtime support

Go runtime can be asked to check for race conditions

```
$ go run -race myapp.go
```

Why not the default?

- "The cost of race detection varies by program, but for a typical program, memory usage may increase by 5-10x and execution time by 2-20x" https://golang.org/doc/articles/race_detector#Runtime_Overheads

Mutex in Go

Import the **sync** package

Create a struct to hold the mutex and variable

Mutex has two methods: **Lock** and **Unlock**

Gorountines wait before accessing the variable until they have the lock

My Better Bank App

```go
import (
	"fmt"
	"strconv"
	"sync"
)

type account struct {
	mu     sync.Mutex
	amount float64
}
```

Locking critical sections

```go
func (acc *account) Deposit(sum float64) {
    acc.mu.Lock()
    acc.amount += sum
    acc.mu.Unlock()
}

func (acc *account) Withdraw(sum float64) {
    acc.mu.Lock()
    acc.amount -= sum
    acc.mu.Unlock()
}
```

Demo: My Better Bank App

Concept Review

Concept review

Go supports complex concurrent systems with compact code

Common concurrency patterns can be elegantly implemented via

- Goroutines
- Channels
- Select
- Timeouts

Yet...there are still practical situations where channels are not suited, and a traditional locking by mutual exclusion is preferred

Recommended Reading

Any Rob Pike talk on Go & Concurrency :-)

- https://www.youtube.com/watch?v=f6kdp27TYZs

Fundamentals of concurrent programming

- Mutual Exclusion lock (mutex)
- https://yourbasic.org/golang/mutex-explained/