# Concurrency Patterns and Mutexes

Ric Glassey
glassey@kth.se

# Outline

- Concurrency Patterns and Mutexes
- Aim: “**Appreciate the expressiveness Go brings to a complex system, but recognise the need for simple solutions also**”
- Google Search:
  - Fan-in pattern
  - Select with Timeout
  - Replication
- Simple Manual Locking with Mutexes

# Concurrency Patterns

# (Toy) Google Search

- Go allows complex concurrent ideas to be expressed with little overhead in code
- Sequence of examples show how concurrency patterns are introduced with relative ease in Go
    - Fan-in pattern
    - Select or Timeout pattern
    - Replication pattern

# Sequential Google

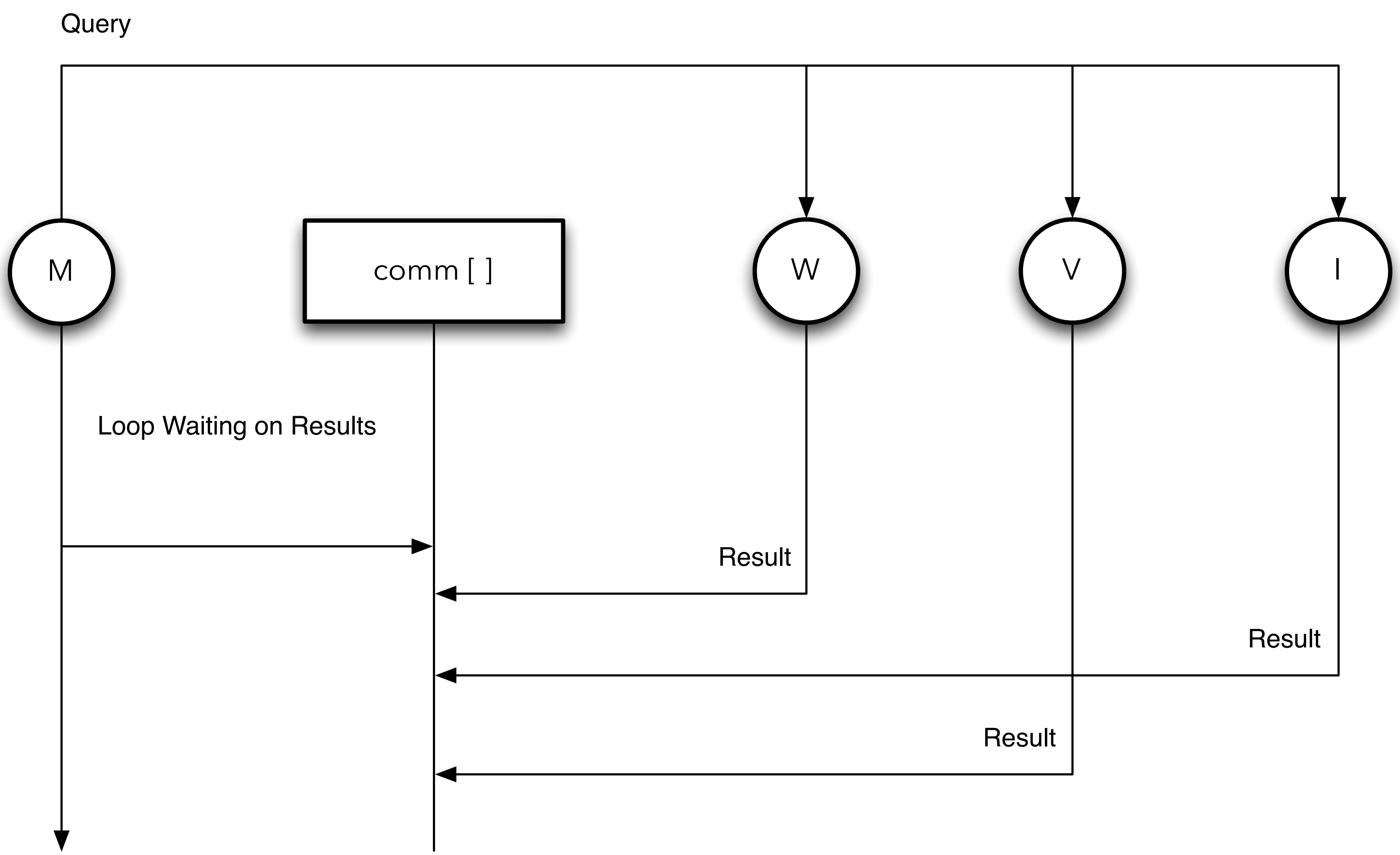
```go
type Result string

type Search func(query string) Result

var (
   Web = fakeSearch("web")
   Image = fakeSearch("image")
   Video = fakeSearch("video")
)

func fakeSearch(kind string) Search {
   return func(query string) Result {
      time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)
      return Result(fmt.Sprintf("%s result for %q\n", kind, query))
   }
}
```

see: https://talks.golang.org/2012/concurrency.slide#1

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

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

see: https://talks.golang.org/2012/concurrency.slide#1

# Fan-in Pattern

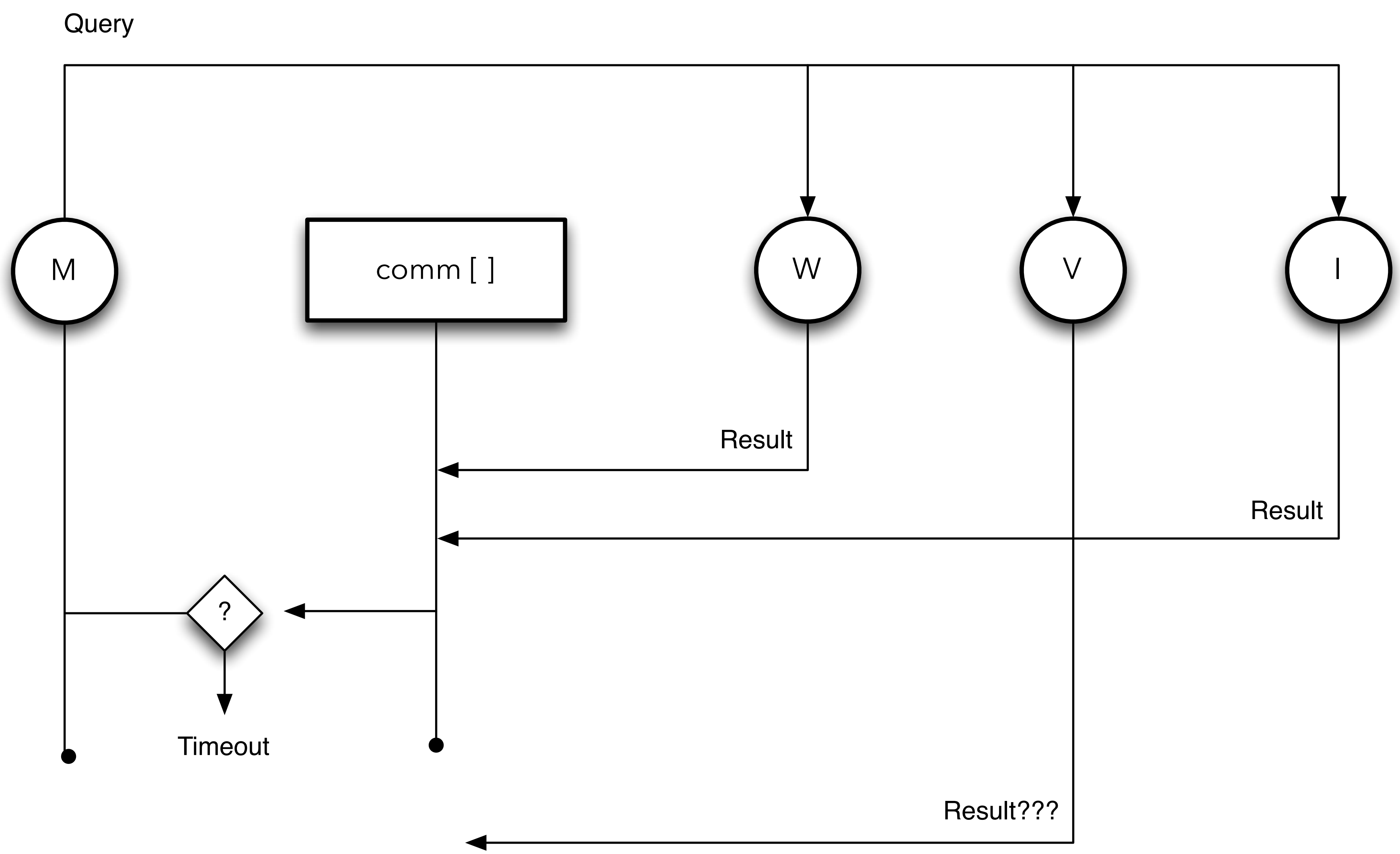



Fan-in Pattern: results are processed as they arrive

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

see: https://talks.golang.org/2012/concurrency.slide#1

# Select or Timeout Pattern

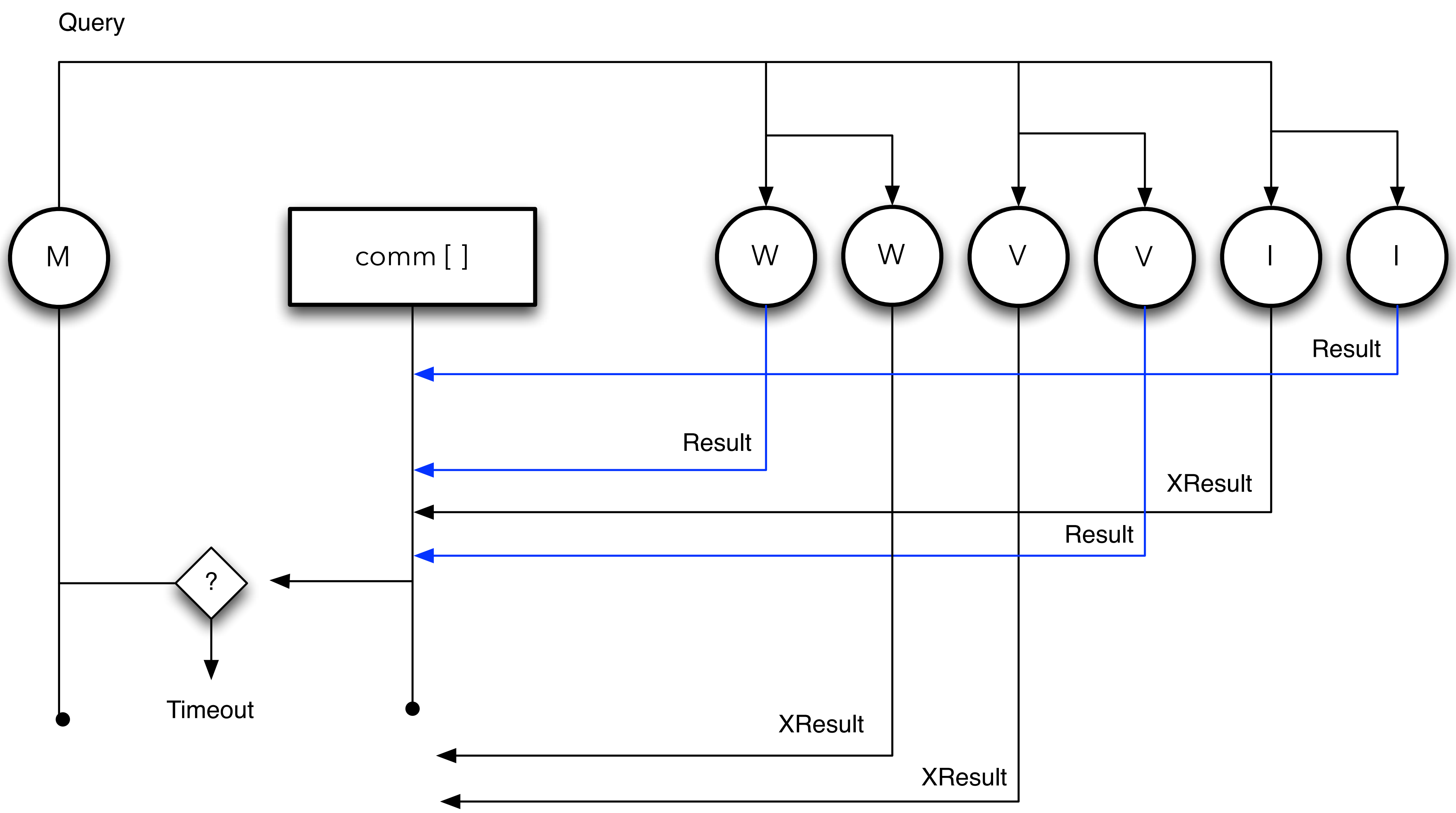



We may not want to wait for slow services

```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(80 * time.Millisecond)
   for i := 0; i < 3; i++ {
      select {
      case result := <- comm:
         results = append(results, result)
      case <- timeout:
         fmt.Println("timed out")
         return
      }
   }
   return
}
```

see: https://talks.golang.org/2012/concurrency.slide#1

# Replication Pattern




Rather than lose results, create many replicas

```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) } ()

   // removed results collection
   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
}
```

see: https://talks.golang.org/2012/concurrency.slide#1

# Complex system; Low cost

- The evolution of the toy google search engine mimics the types of systems required today
  - fast
  - progressive
  - redundancy
- Yet there has been very little code used

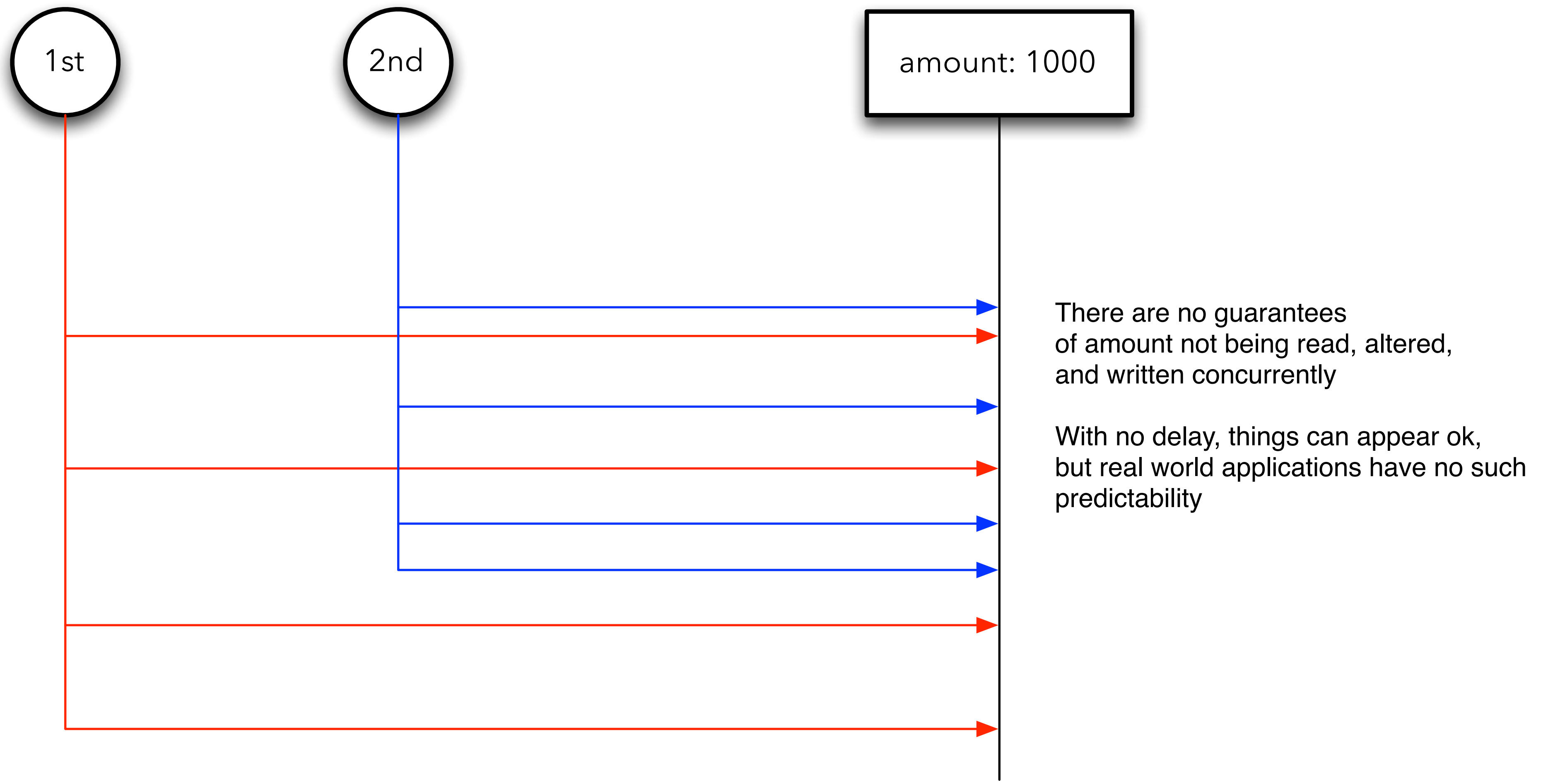
# Locking by Mutex

# Simple is better

- Despite powerful concurrency, synchronisation and communication constructs, some situations are very simple
    - e.g. lock a single variable from concurrent access
- **Mutual exclusion** ensures that no two threads are in their critical section at the same time
    - e.g. reading, updating and writing to a shared variable

# Using a Mutex

- A **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 threads




# Banking application - potential race condition

```go
type account struct {
    amount    float64
}

func (acc *account) Deposit(sum float64) {
    // The bank clerk is randomly slow/fast
    time.Sleep(time.Duration(rand.Int31n(500)) * time.Millisecond)
    acc.amount += sum
}

func (acc *account) Withdraw(sum float64) {
    // Cash machines are a little slow
    time.Sleep(time.Duration(rand.Int31n(500)) * time.Millisecond)
    acc.amount -= sum
}

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

```go
func main() {
   // remember to seed the random number generator
   rand.Seed( time.Now().UTC().UnixNano())

   var joint_account account
    joint_account.Deposit(1000.00)
    fmt.Println(joint_account.Balance())

   // stop main from quitting before threads
   comm := make(chan bool)

    // Two people are accessing the account concurrently
   // In total they deposit 300 and withdraw 400
   // We expect the final balance to be 900

   go func () {
        joint_account.Deposit(50.00)
        joint_account.Deposit(50.00)
        joint_account.Withdraw(200.00)
        joint_account.Deposit(50.00)
    }()

    go func () {
        joint_account.Deposit(50.00)
        joint_account.Deposit(50.00)
        joint_account.Withdraw(200.00)
        joint_account.Deposit(50.00)
        comm<-true
    }()

    <-comm
    fmt.Println(joint_account.Balance())
}
```

# Detecting Race Conditions

# Runtime Detection

- go run **-race** code.go
- Will attempt to detect some race conditions and report potential regions of concern

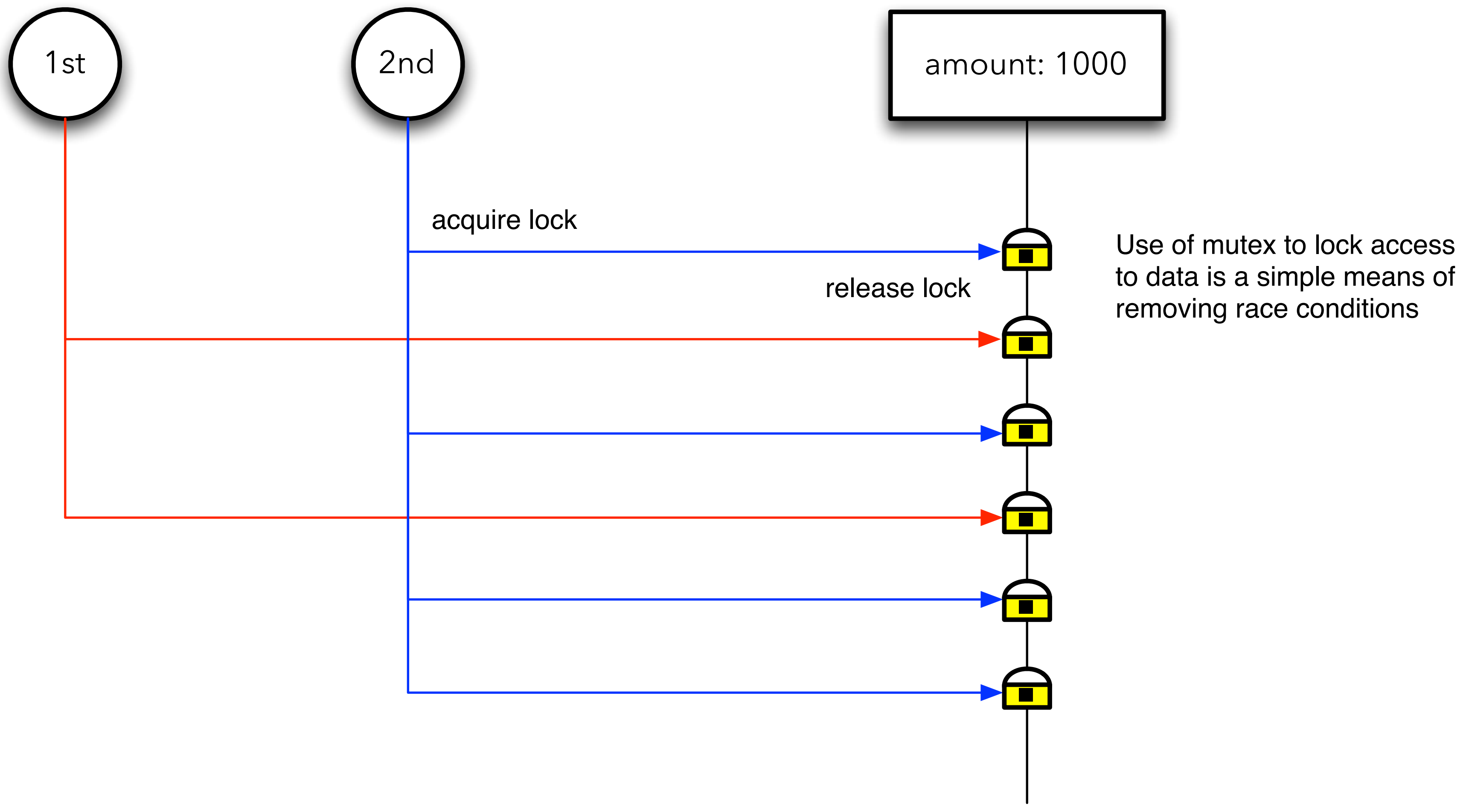
```
==================
   WARNING: DATA RACE
   Read by goroutine 7:
     main.(*account).Deposit()
         /Users/ric/Dropbox/kth/teaching/dd1339-java/content/lecture-09p/
resources/banking-race/banking-race.go:17 +0x59
     main.func·002()
         /Users/ric/Dropbox/kth/teaching/dd1339-java/content/lecture-09p/
resources/banking-race/banking-race.go:57 +0x50

   Previous write by goroutine 6:
     main.(*account).Deposit()
         /Users/ric/Dropbox/kth/teaching/dd1339-java/content/lecture-09p/
resources/banking-race/banking-race.go:17 +0x71
     main.func·001()
         /Users/ric/Dropbox/kth/teaching/dd1339-java/content/lecture-09p/
resources/banking-race/banking-race.go:49 +0x47
```

# Mutex in Go

- Run the banking example enough times…
  - 850, 900, 1050…
- Sync package provides Mutex type
  - This package is mostly for low-level concerns
  - But Mutex is a simple solution to a common problem
  - Use of channels would increase complexity and overhead and lose elegance

# Mutex in Go

- Mutex type has two methods:
  - Lock
  - Unlock
- Go routines not having the lock cannot access the mutex until it has been unlocked
- Common pattern in Go is to use a struct to encompass the variable and mutex




amount is now protected by locking

```go
import (
    "fmt"
    "strconv"
    "math/rand"
    "time"
    "sync"
)

type account struct {
    mu        sync.Mutex
    amount    float64
}

func (acc *account) Deposit(sum float64) {
    // The bank clerk is randomly slow/fast
    acc.mu.Lock()
    time.Sleep(time.Duration(rand.Int31n(250)) * time.Millisecond)
    acc.amount += sum
    acc.mu.Unlock()
}

func (acc *account) Withdraw(sum float64) {
    // Cash machines are a little slow also
    acc.mu.Lock()
    time.Sleep(time.Duration(rand.Int31n(250)) * time.Millisecond)
    acc.amount -= sum
    acc.mu.Unlock()
}
```

# Summary

- Go supports complex concurrent systems at a low cost to the developer
- Yet…there are occasions where channels are not suited, and a traditional mutex is more elegant

# Reading

- Fundamentals of concurrent programming
  - by S. Nilsson
  - Required Reading
  - Sections 6 - 8
  - http://www.nada.kth.se/~snilsson/concurrency
- Any Rob Pike talk on Go :-)
  - In particular: https://www.youtube.com/watch?v=f6kdp27TYZs