Understanding Real-World Concurrency Bugs in Go

Tengfei Tu[1], Xiaoyu Liu[2], ***Linhai Song[1]***, and Yiying Zhang[2]

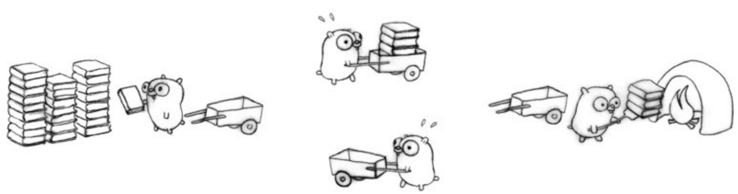
[1]Pennsylvania State University

[2]Purdue University

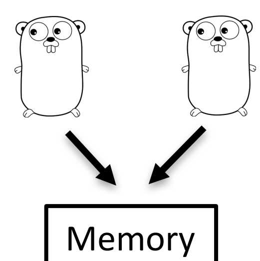
Golang

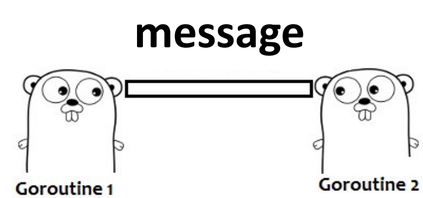
- A young but widely-used programming lang.

- Designed for efficient and *reliable* concurrency
  - Provide lightweight threads, called goroutines
  - Support both message passing and shared memory

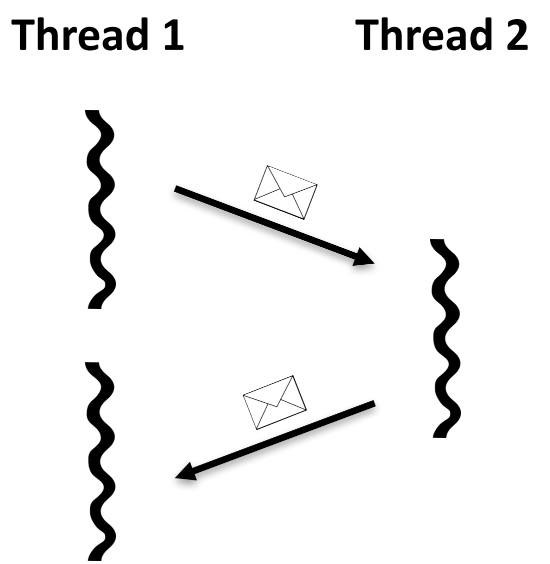


Lightweight threads (goroutines)

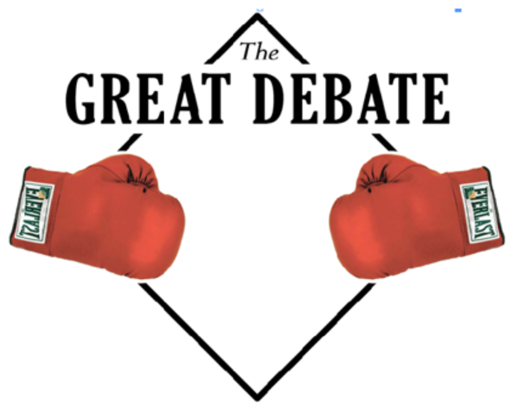


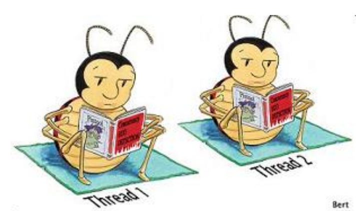


Massage Passing vs. Shared Memory

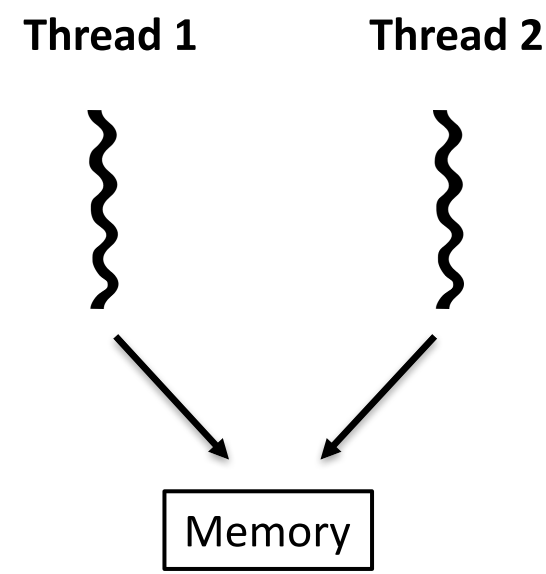


Message Passing





Concurrency Bug



Shared Memory

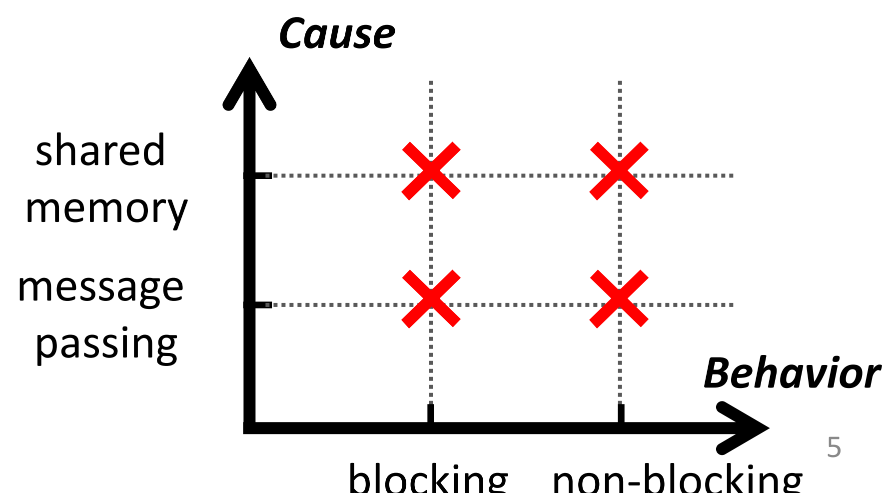
Does Go Do Better?

- Message passing better than shared memory?
- How well does Go prevent concurrency bugs?

The 1st Empirical Study

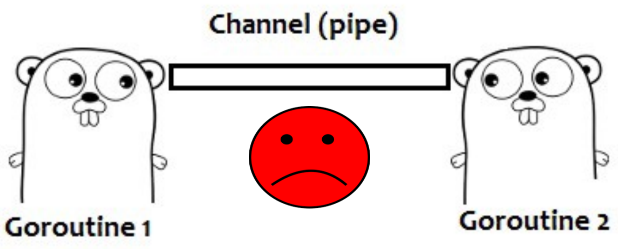
- Collect 171 Go concurrency bugs from 6 apps
  - through manually inspecting GitHub commit log
- How we conduct the study?
  - Taxonomy based on two orthogonal dimensions
    - Root causes and fixing strategies
  - Evaluate two built-in concurrency bug detectors

171 Real-World Go Concurrency Bugs



Highlighted Results

- Message passing can make a lot of bugs
  - sometimes even more than shared memory
- 9 observations for developers' references
- 8 insights to guide future research in Go



Outline

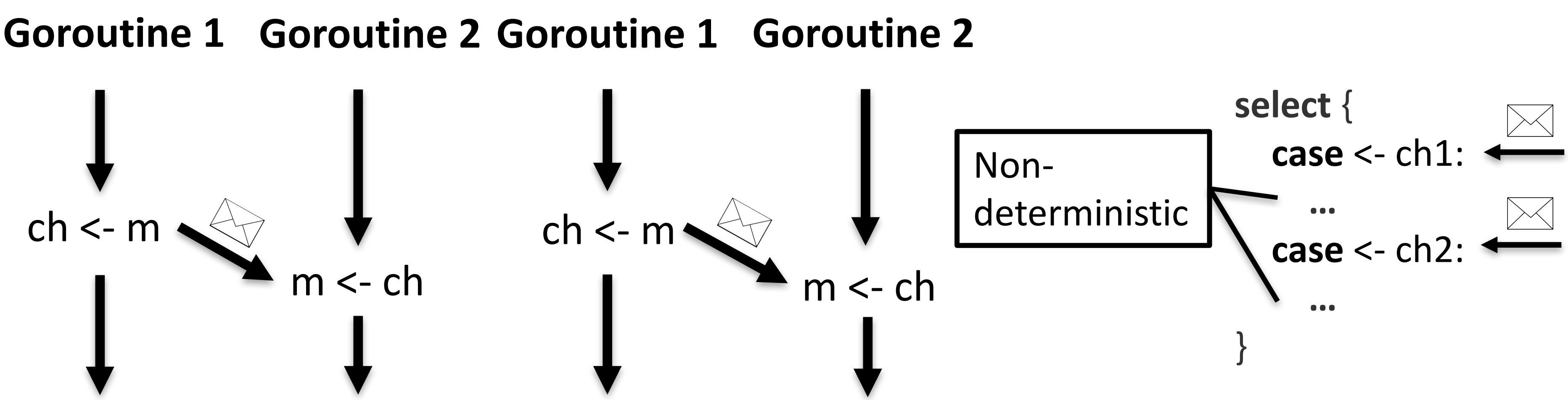
- Introduction
- A real bug example
- Go concurrency bug study
  - Taxonomy
  - Blocking Bug
  - Non-blocking Bug
- Conclusions

Outline

- Introduction
- A real bug example
- Go concurrency bug study
  - Taxonomy
  - Blocking Bug
  - Non-blocking Bug
- Conclusions

Message Passing in Go

- How to pass messages across goroutines?
  - Channel: unbuffered channel vs. buffered channel
  - Select: waiting for multiple channel operations



unbuffered channel **buffered channel** **select**

An Example of Go Concurrency Bug

Parent Goroutine

```
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func()
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```

kubernetes

An Example of Go Concurrency Bug

Parent Goroutine

```
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func()
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```

kubernetes

An Example of Go Concurrency Bug

Parent Goroutine

```
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func()
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```

Child Goroutine

```
go func()
    result := fn()
    ch <- result
}()
```

kubernetes

An Example of Go Concurrency Bug

Parent Goroutine

```
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func()
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```

timeout
signal

Child Goroutine

```
go func()
    result := fn()
    ch <- result
}()
```

blocking and
goroutine leak

kubernetes

An Example of Go Concurrency Bug

Parent Goroutine

```
func finishRequest(t sec) r object {
  ch := make(chan object , 1)
  go func()
    result := fn()
    ch <- result
  }()
  select {
    case result = <- ch:
      return result
    case <- time.timeout(t):
      return nil
  }
} //Kubernetes#5316
```

Child Goroutine

```
go func()
  result := fn()
  ch <- result
}()
```

not blocking any more

kubernetes

New Concurrency Features in Go

```
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func()
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```

anonymous function

buffered channel vs. **unbuffered** channel

use **select** to wait for multiple channels

New Concurrency Features in Go

```
func finishRequest(t sec) r object {
    ch := make(chan object)
    go func()
        result := fn()
        ch <- result
    }()
    select {
        case result = <- ch:
            return result
        case <- time.timeout(t):
            return nil
    }
} //Kubernetes#5316
```

anonymous function

buffered channel vs. **unbuffered** channel

use **select** to wait for multiple channels

Outline

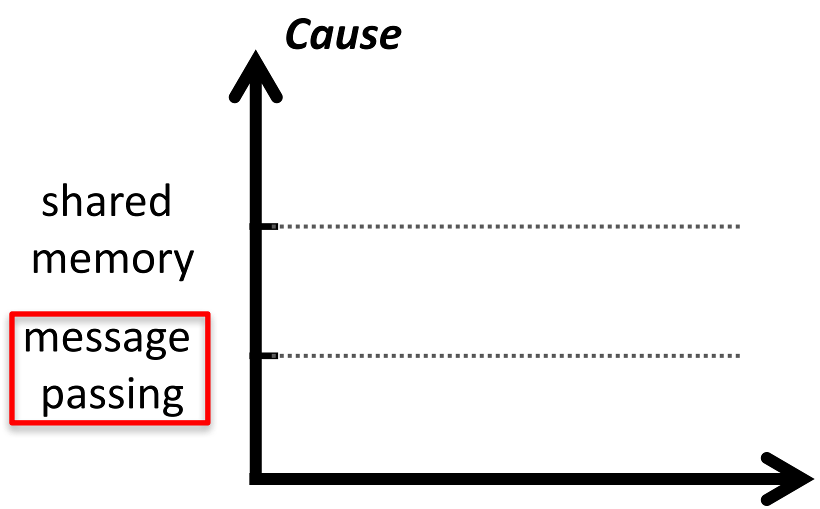
- Introduction
- A real bug example
- Go concurrency bug study
  - Taxonomy
  - Blocking Bug
  - Non-blocking Bug
- Conclusions

Bug Taxonomy

- Categorize bugs based on two dimensions
  - Root cause: shared memory vs. message passing

```
func finishRequest(t sec) r object {
  ch := make(chan object)
  go func()
    result := fn()
    ch <- result
  }()
  select {
    case result = <- ch:
      return result
    case <- time.timeout(t):
      return nil
  }
} //Kubernetes#5316
```

channel

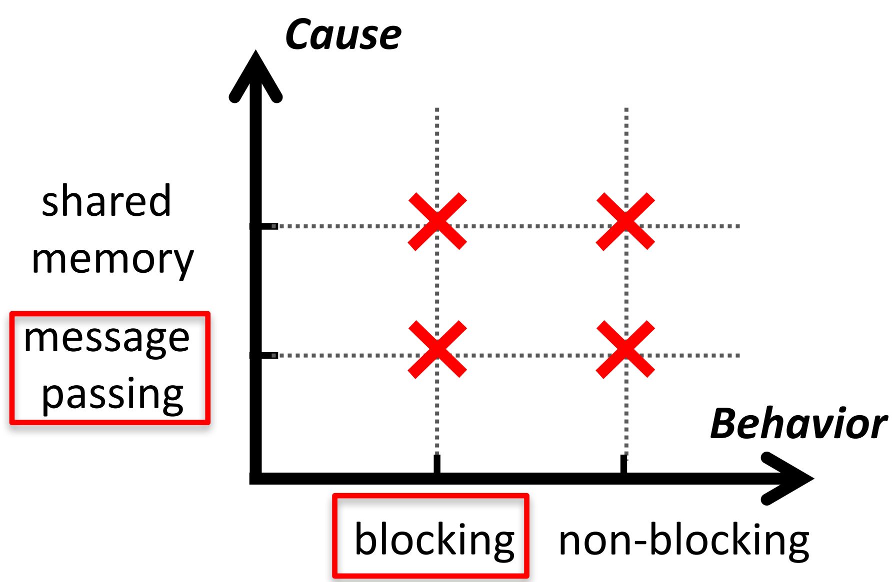


Bug Taxonomy

- Categorize bugs based on two dimensions
  - Root cause: shared memory vs. message passing
  - Behavior: blocking vs. non-blocking

```
func finishRequest(t sec) r object {
   ch := make(chan object)
   go func()
      result := fn()
      ch <- result
   }()
   select {
      case result = <- ch:
         return result
      case <- time.timeout(t):
         return nil
   }
} //Kubernetes#5316
```

blocking



Bug Taxonomy

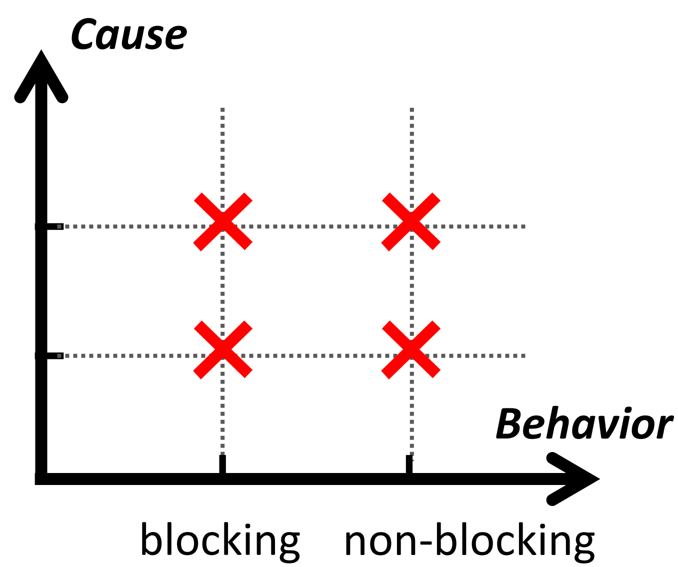
- Categorize bugs based on two dimensions
  - Root cause: shared memory vs. message passing
  - Behavior: blocking vs. non-blocking

Cause	Behavior	
	blocking	non-blocking
shared memory	36	69
message passing	49	17

105 shared memory

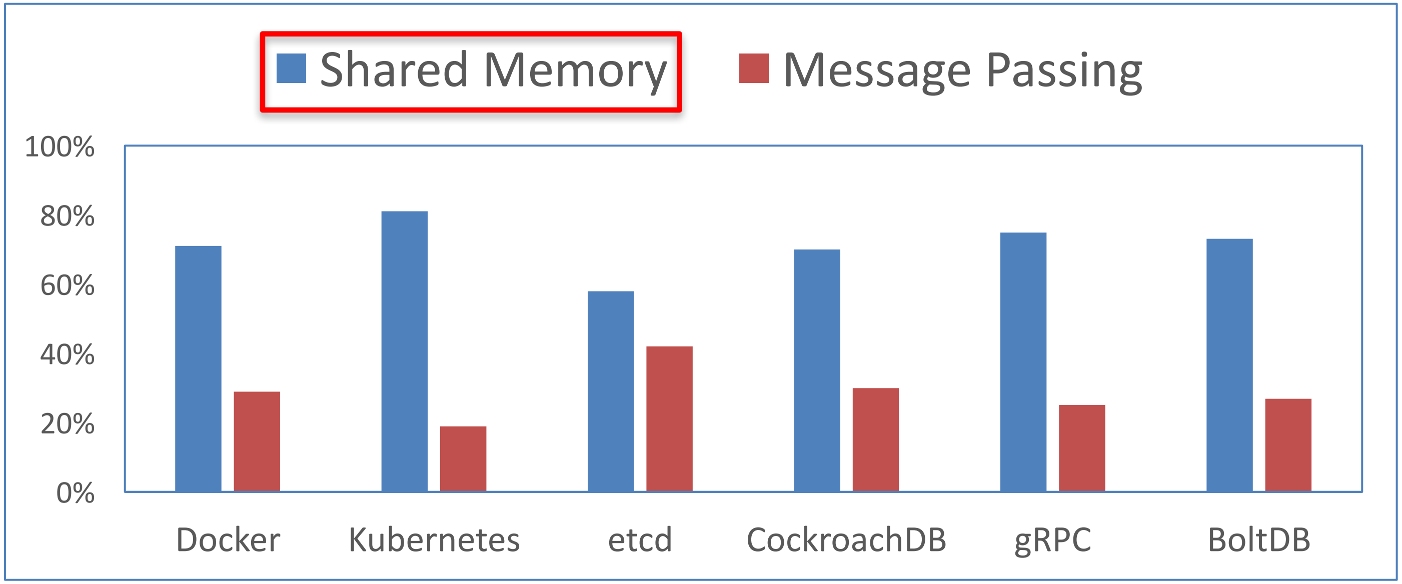
66 message passing

85 86



Concurrency Usage Study

Observation: Share memory synchronizations are used more often in Go applications.

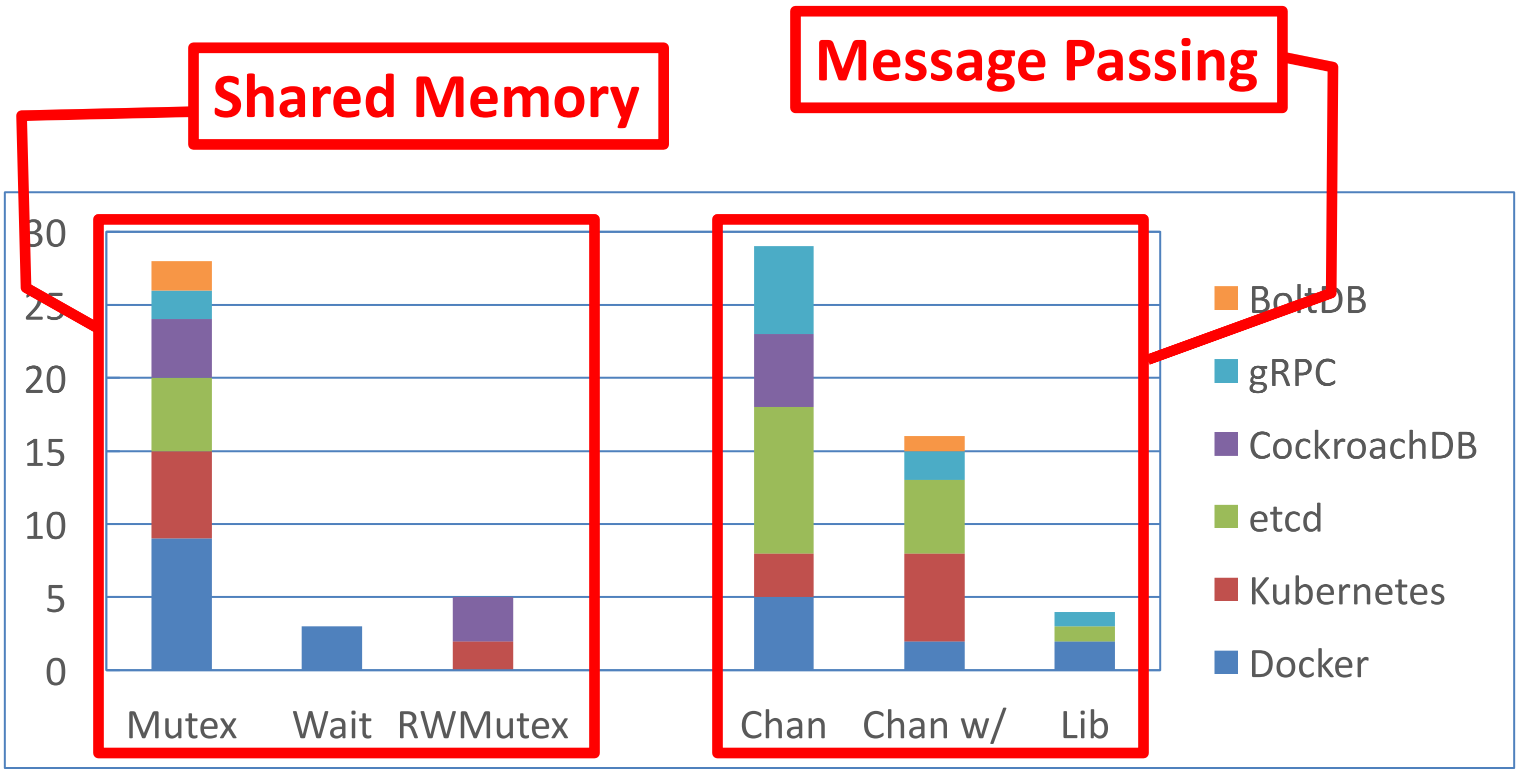


Outline

- Introduction
- A real bug example
- Go concurrency bug study
  - Taxonomy
  - Blocking Bug
  - Non-blocking Bug
- Conclusions

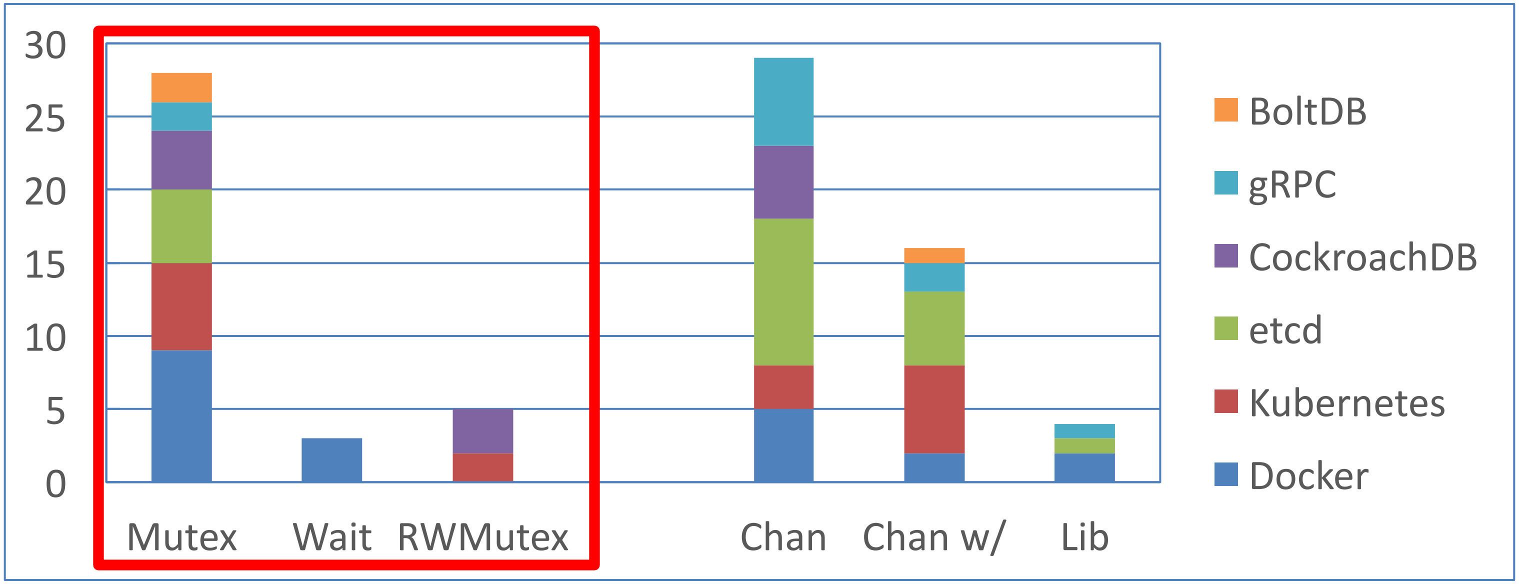
Root Causes

- Conducting blocking operations
  - to protect shared memory accesses
  - to pass message across goroutines

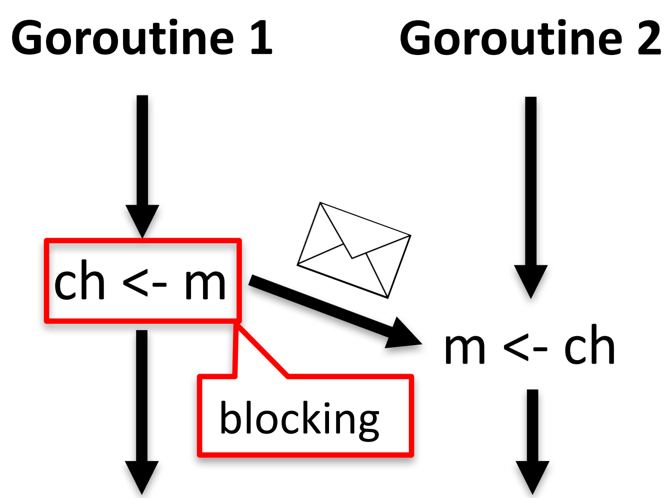


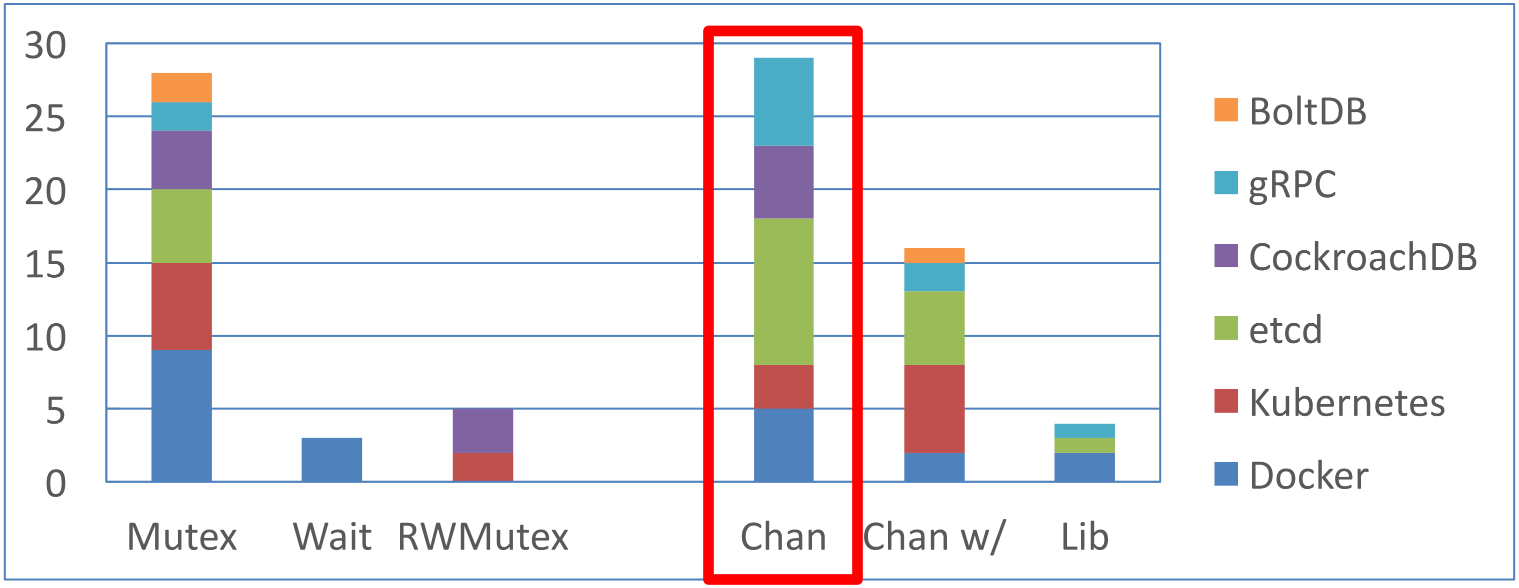
(mis)Protecting Shared Memory

Observation: Most blocking bugs caused by shared memory synchronizations have the same causes as traditional languages.



Misuse of Channel

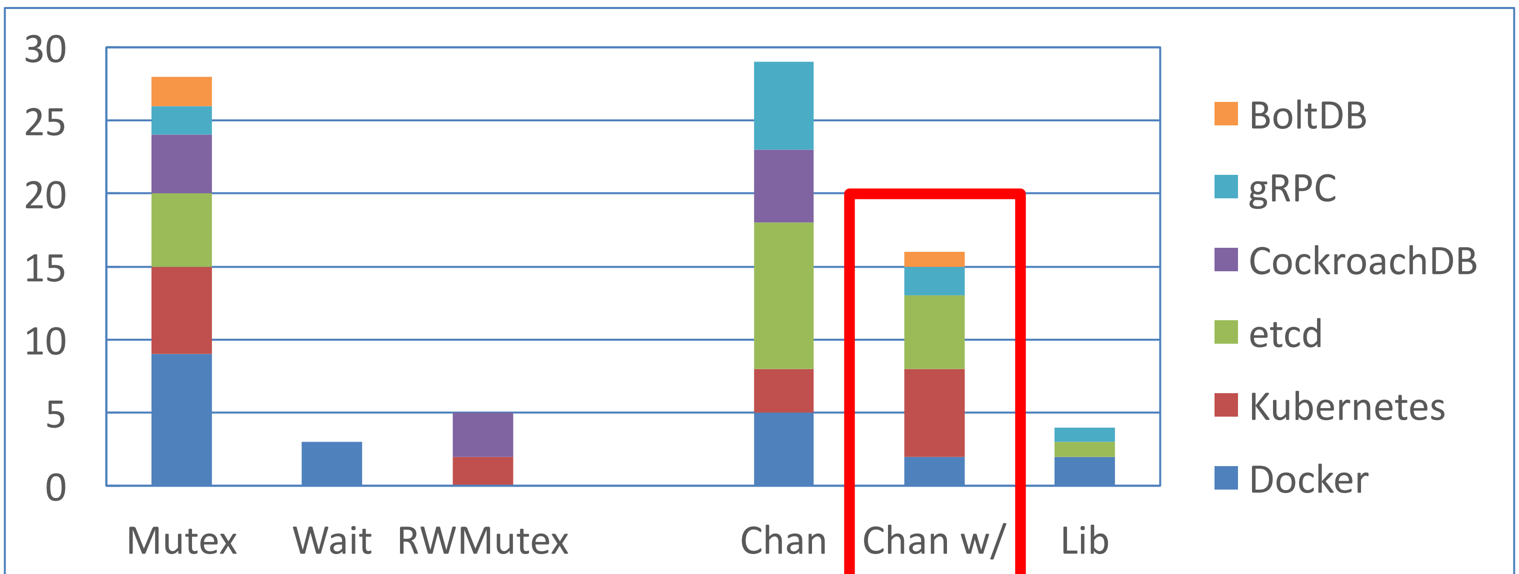




Misuse of Channel with Lock

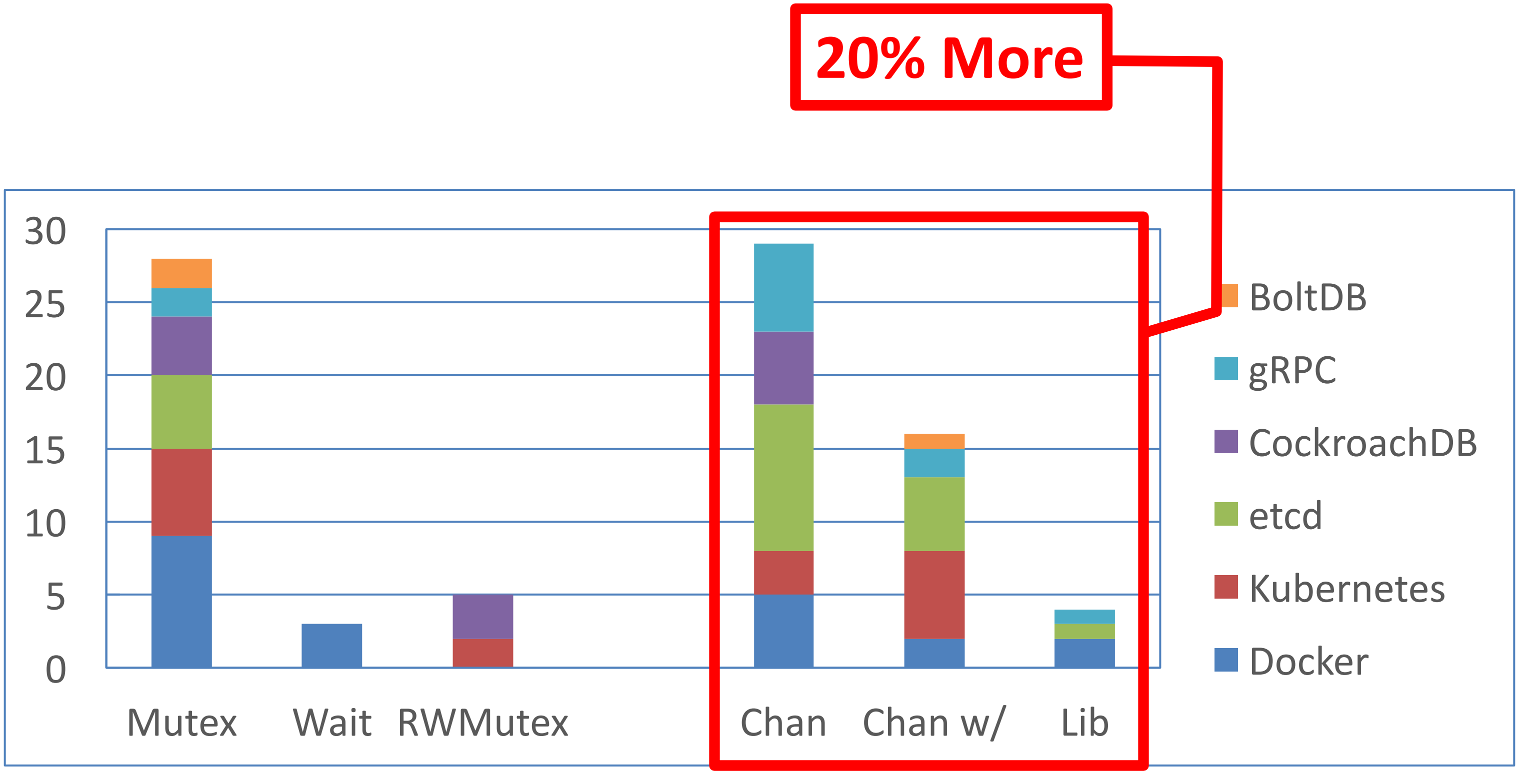
```
func goroutine1() {
    m.Lock()
    ch <- request

    m.Unlock()
}
```

```
func goroutine2() {
    for {
        m.Lock()
        m.Unlock()
        request <- ch
    }
}
```



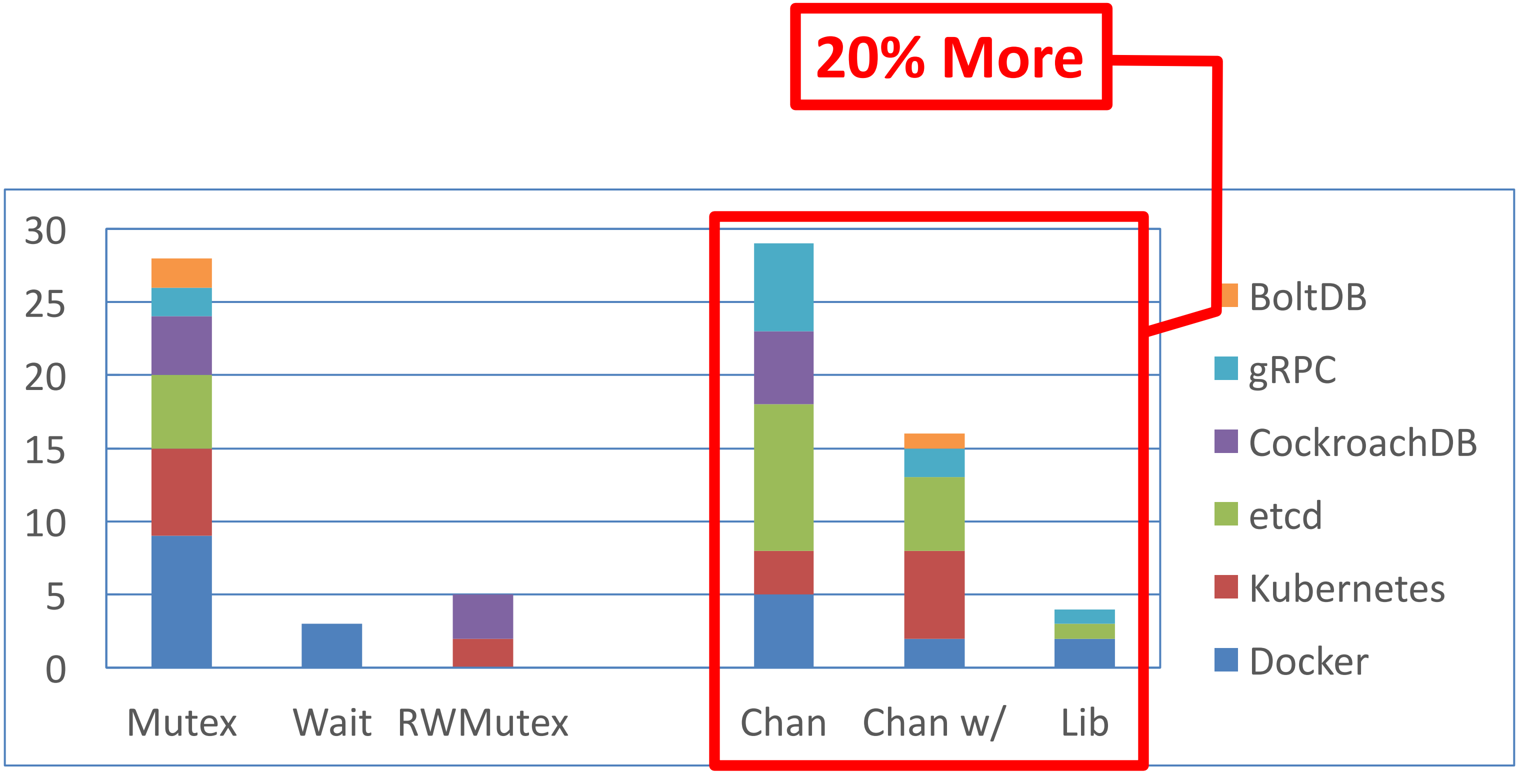
Observation

Observation: more blocking bugs in our studied Go applications are caused by wrong message passing.



Implication

Implication: we call for attention to the potential danger in programming with message passing.

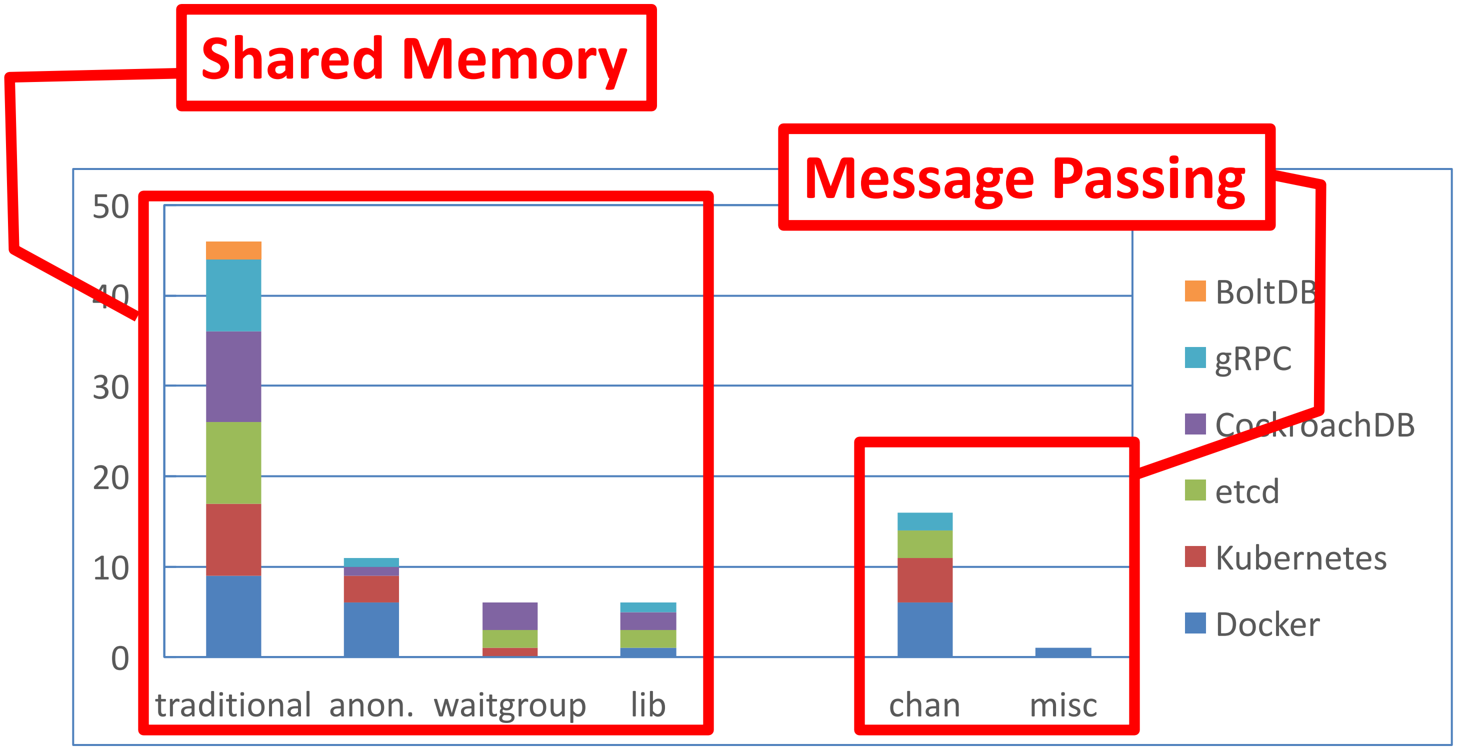


Outline

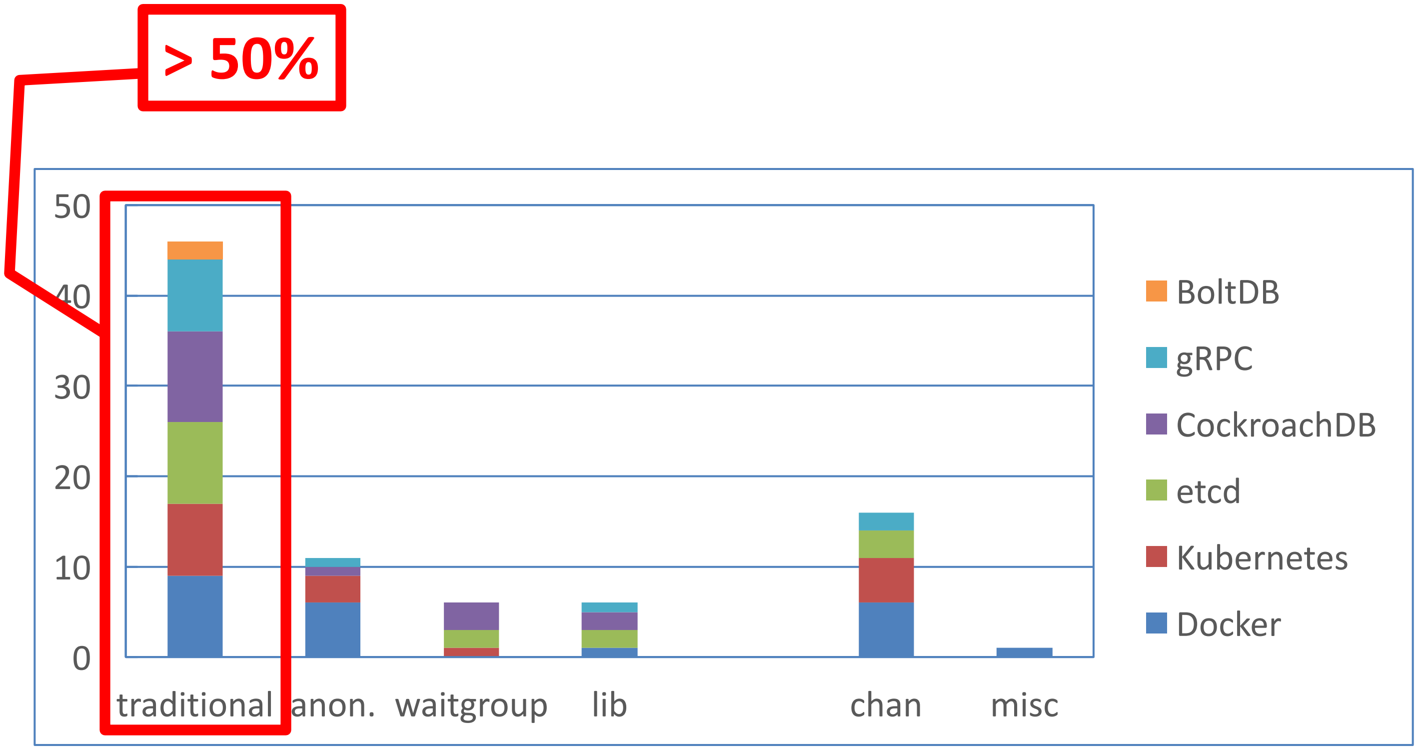
- Introduction
- A real bug example
- Go concurrency bug study
  - Taxonomy
  - Blocking Bug
  - Non-blocking Bug
- Conclusions

Root Causes

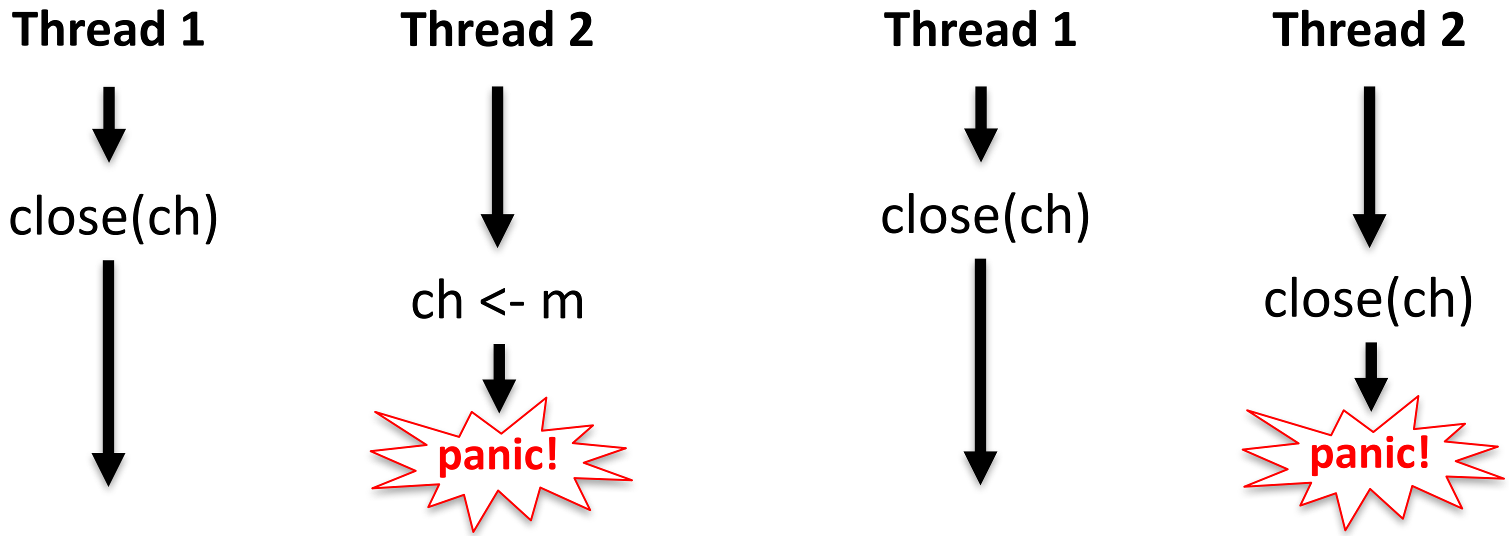
- Failing to protect shared memory
- Errors during message passing

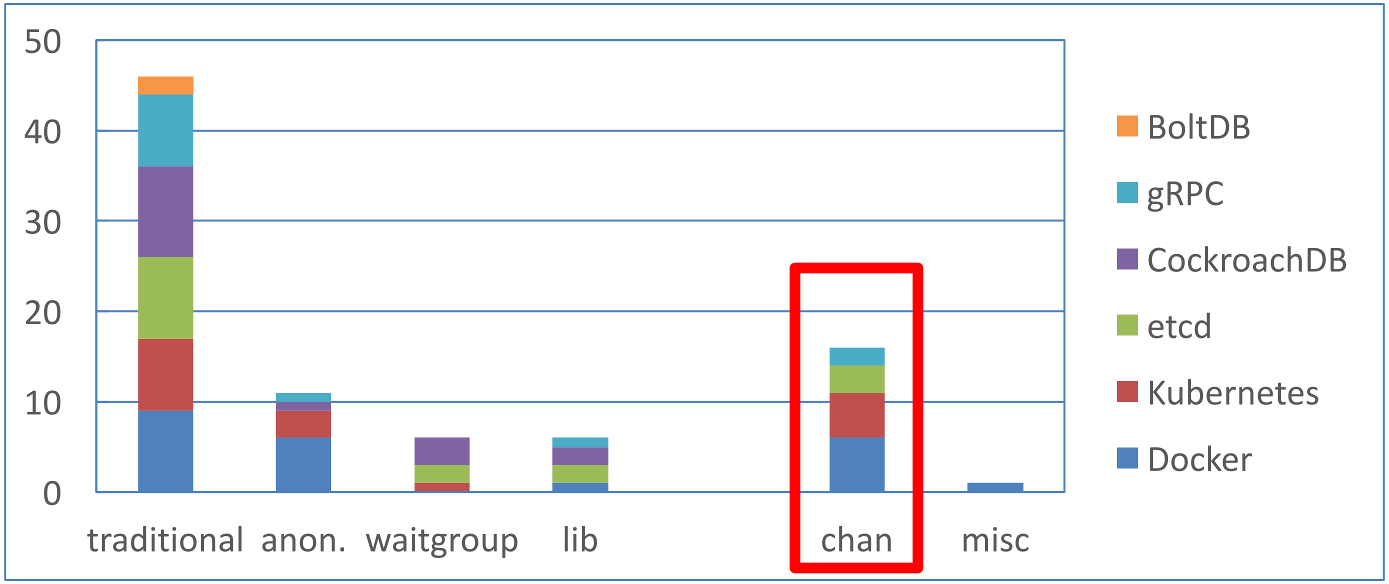


Traditional Bugs



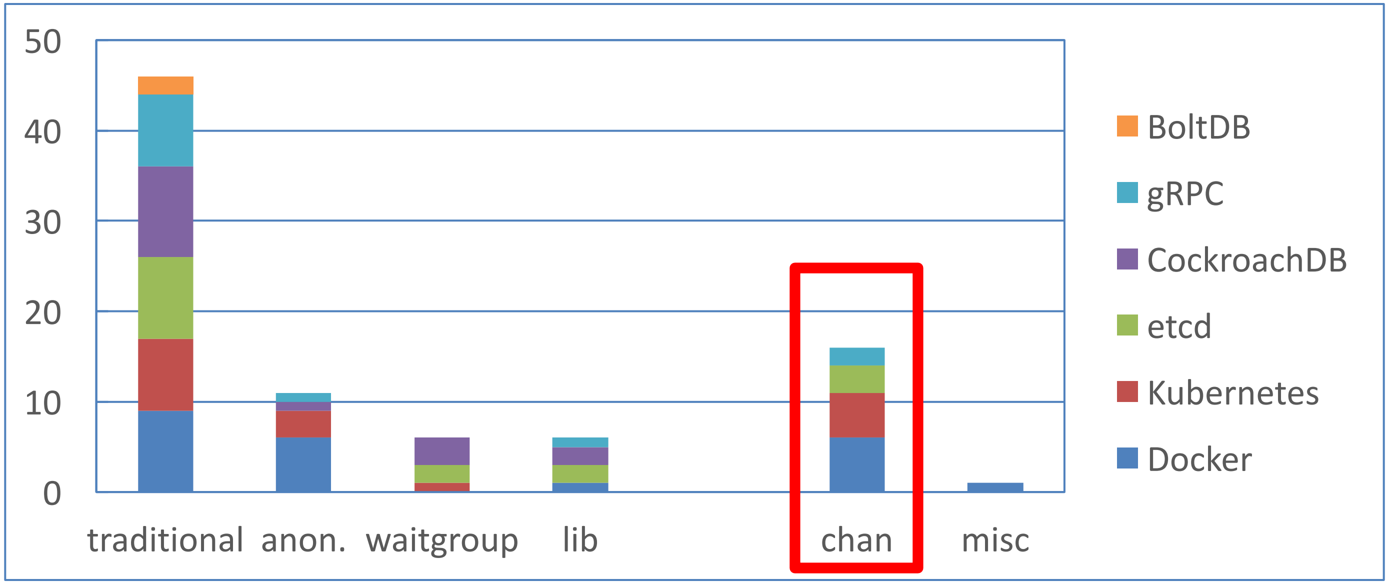
Misusing Channel





Implication

Implication: new concurrency mechanisms Go introduced can themselves be the reasons of more concurrency bugs.



Conclusions

- 1st empirical study on go concurrency bugs
  - shared memory vs. message passing
  - blocking bugs vs. non-blocking bugs

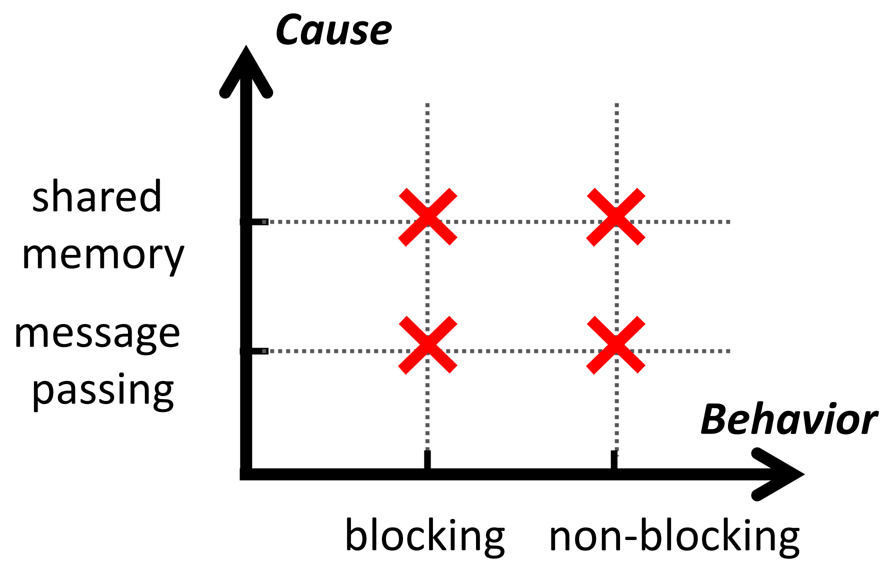
  paper contains more details (contact us for more)
- Future works
  - Statically detecting go concurrency bugs
    - checkers built based on identified buggy patterns
    - Already found concurrency bugs in real applications

Thanks a lot!

Questions?

docker kubernetes GRPC
etcd BlotDB
Cockroach DB

171 Real-World Go Concurrency Bugs



Data Set: https://github.com/system-pclub/go-concurrency-bugs