bit.ly/gogetth-sync-workers

Distributed workers In Golang

Synchronization Between workers In Golang

WISESIGHT

Lattapon Yodsuwan
(Lat)

Napon Mekavuthikul
(Bone)

Outline

- Basic concurrency
  - Goroutine
  - Channel
  - Select
- Fan-in pattern
  - Multi queue consumer
- The “ลุงประหยัด” problem
  - 1 worker vs multiple workers
  - Broadcasting msg with Rabbitmq Publish/Subscribe (fan-out)
  - Cancelling goroutines by closing channel

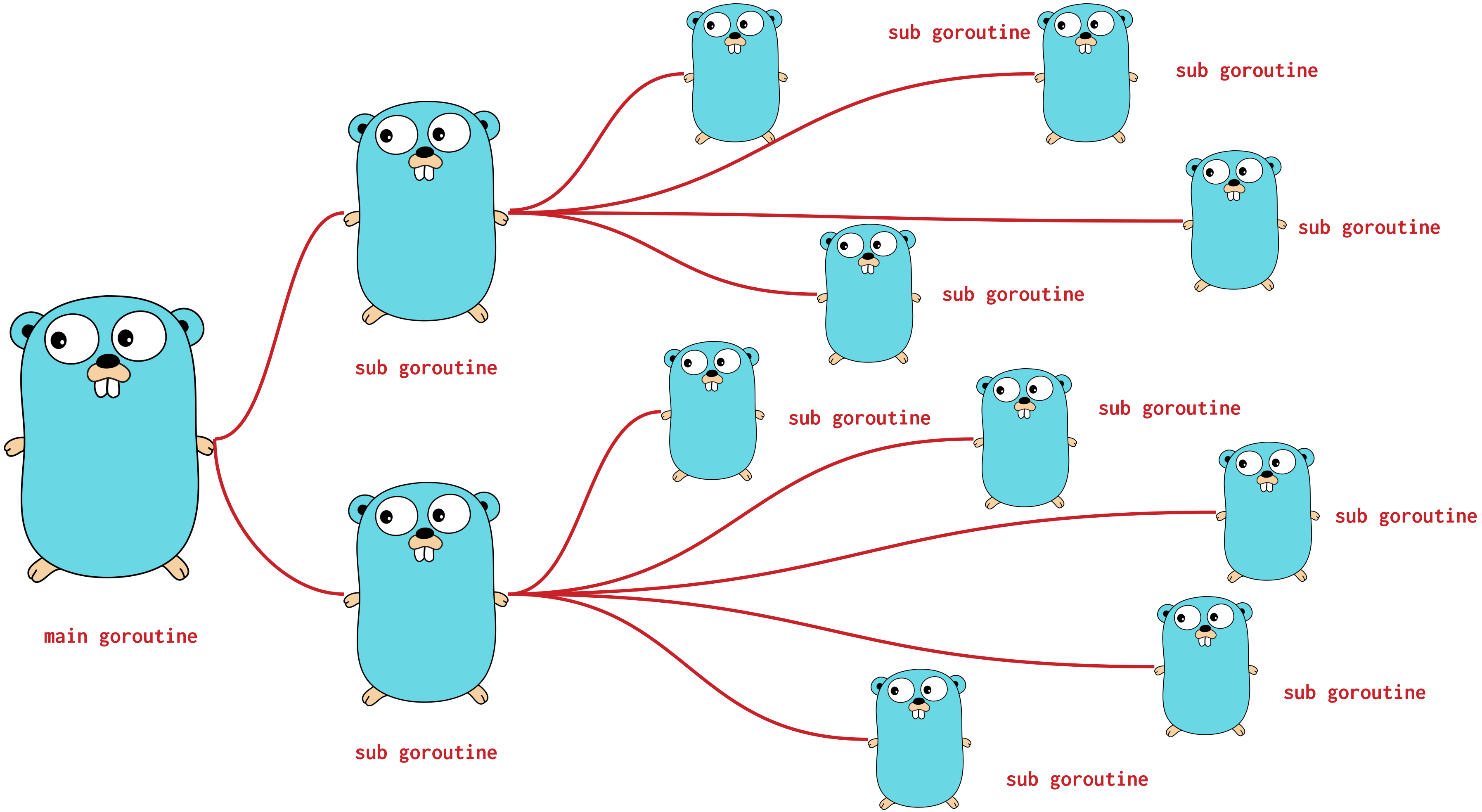
Concurrency

➔ Goroutines

➔ Channels

➔ Select

Goroutines


sub goroutine
sub goroutine
sub goroutine
sub goroutine
sub goroutine
sub goroutine
sub goroutine
main goroutine
sub goroutine
sub goroutine
sub goroutine
sub goroutine

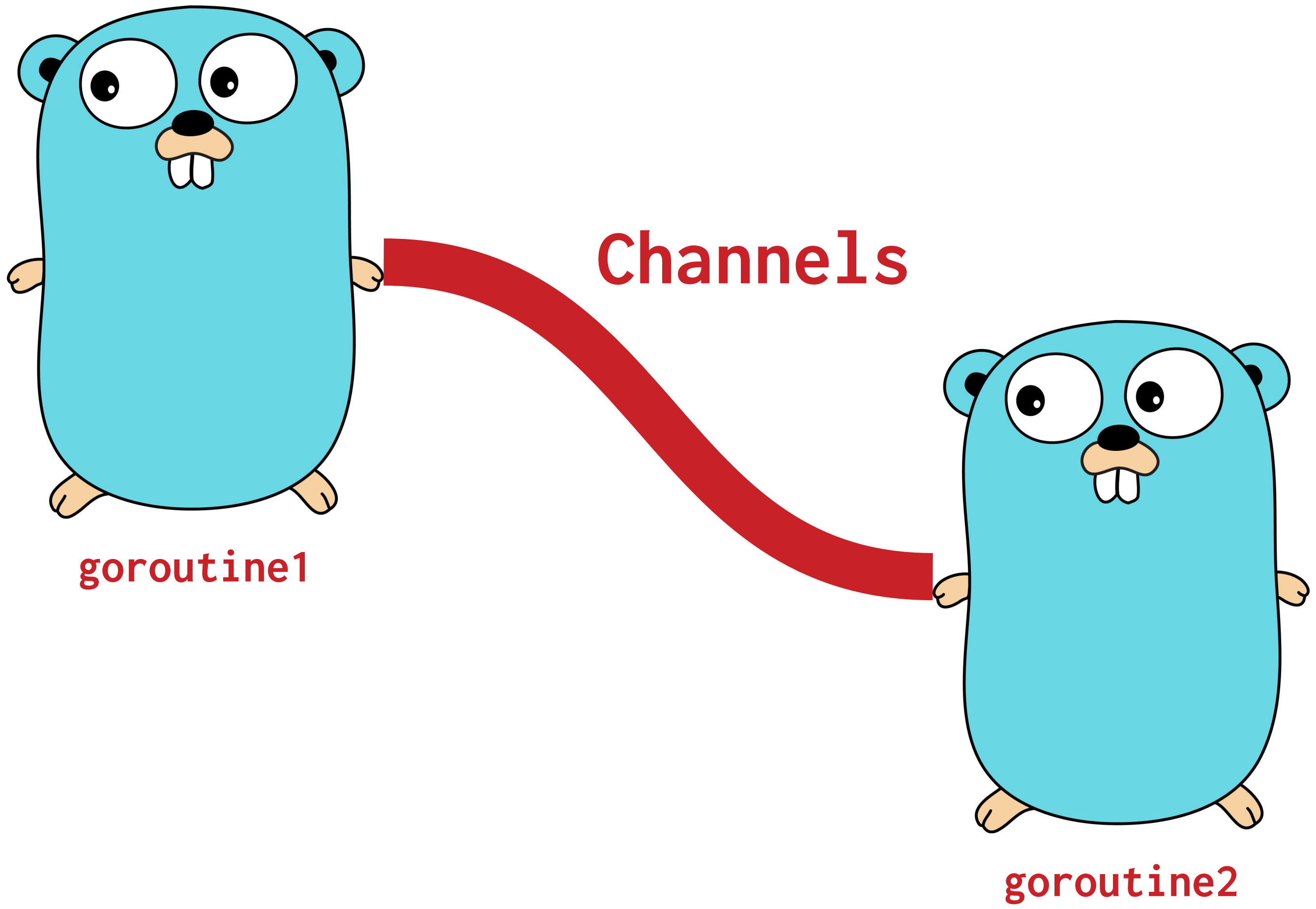
"go"

for new goroutine

Goroutines

```
package main

import (...)

func wait(wg) {
	defer wg.Done()
	time.Sleep(time.Second * 1)
	fmt.Println("Wait for 1 sec")
}

func main() {
	var wg sync.WaitGroup
	wg.Add(1)
	go wait(wg) // Run new routine
	wg.Wait()
}
```

Channels


Channels
goroutine1
goroutine2

Channels

```
package main

import (...)

func wait(n int, c chan string) {
      time.Sleep(time.Second * 1)
      c <- fmt.Sprintf("Number %d", n)
}

func main() {
      s := [...]int{1, 2, 3, 4, 5, 6}
      c := make(chan string)
      for n := range s {
            go wait(n, c)
      }
      for i := 0; i < len(s); i++ {
            fmt.Println(<-c)
      }
}
```

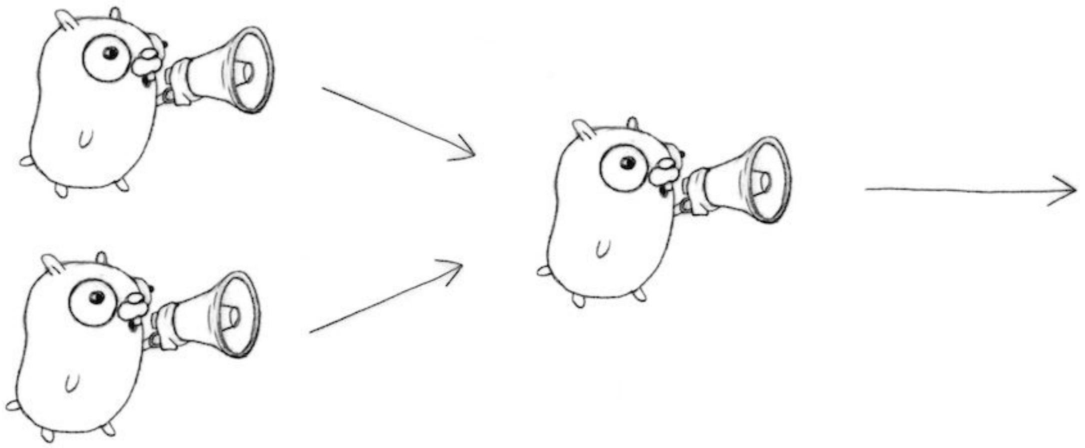
Channels (close)

```
package main

import (...)

func wait(n int, c chan string) {
	time.Sleep(time.Second * 1)
	c <- fmt.Sprintf("Number %d", n)
}
```

```
func main() {
	s := [...]int{1, 2, 3, 4, 5, 6}
	c := make(chan string)
	for n := range s {
		go wait(n, c)
	}
	for i := 0; i < 3; i++ {
		fmt.Println(<-c)
	}
	close(c)
}
```

Select

Select



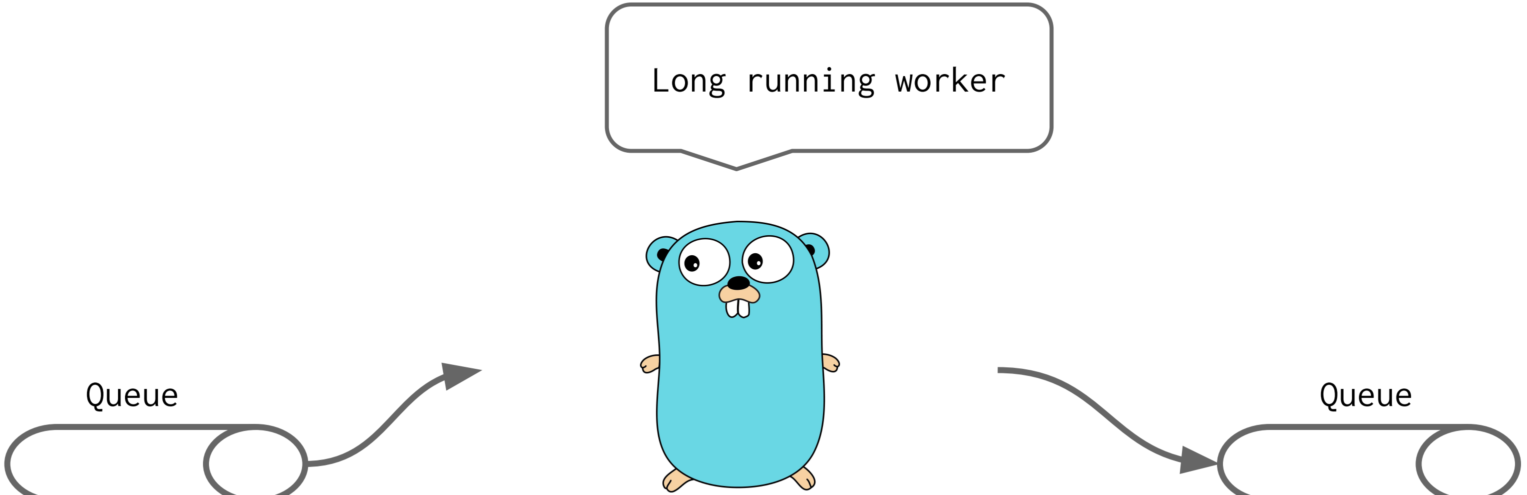
Select

```
package main

import (...)

func fibonacci(c, quit chan int) {
	x, y := 0, 1
	for {
		select {
		case c <- x:
			x, y = y, x+y
		case <-quit:
			fmt.Println("quit")
			return
		}
	}
}
```

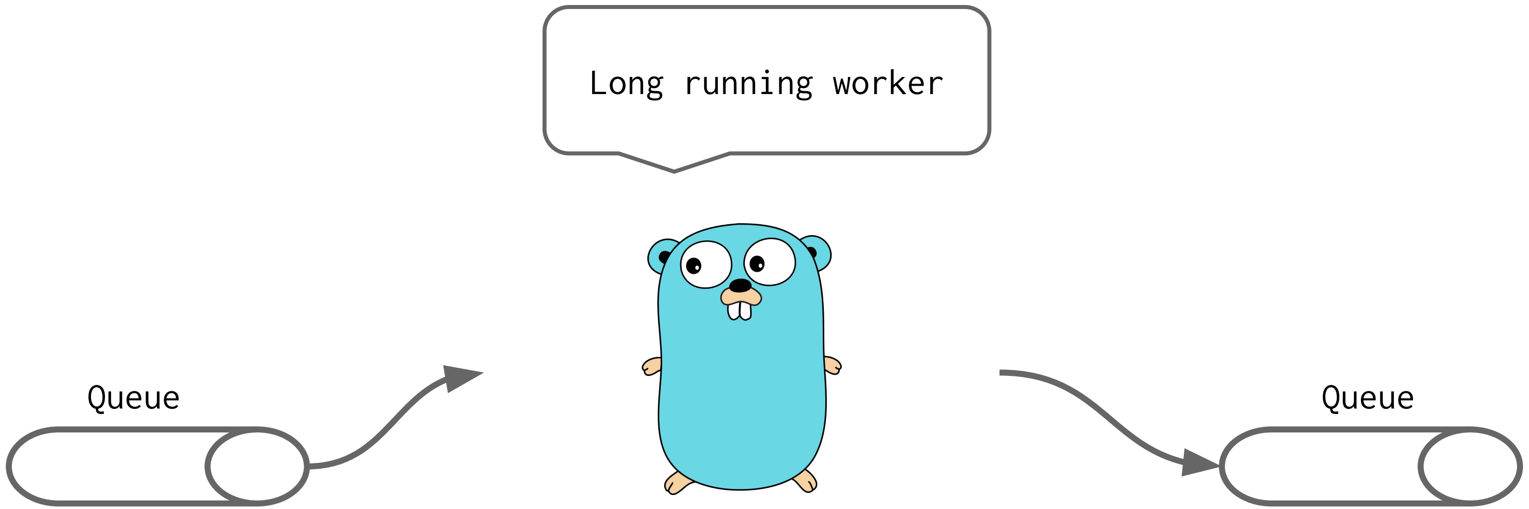
```
func main() {
	c := make(chan int)
	quit := make(chan int)
	go func() {
		for i := 0; i < 10; i++ {
			fmt.Println(<-c)
		}
		quit <- 0
	}()

	fibonacci(c, quit)
}
```

Example use case

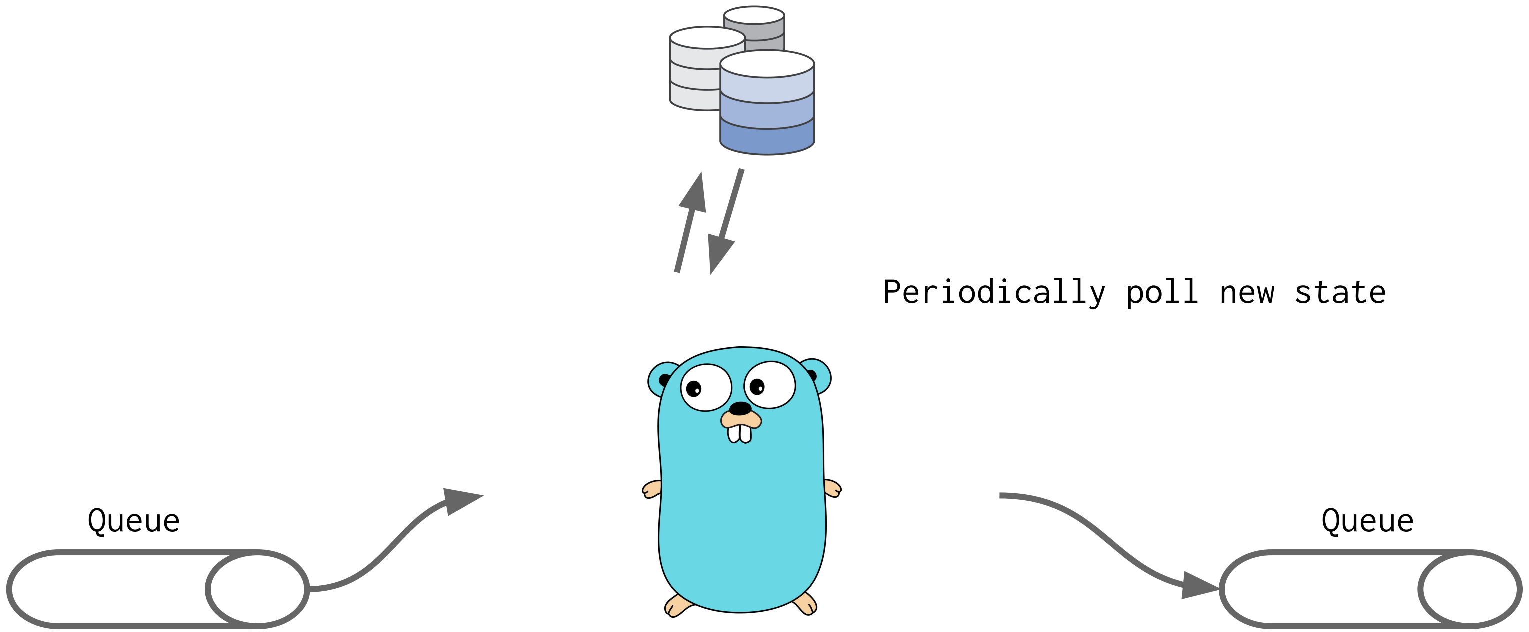
The worker use case



The worker use case



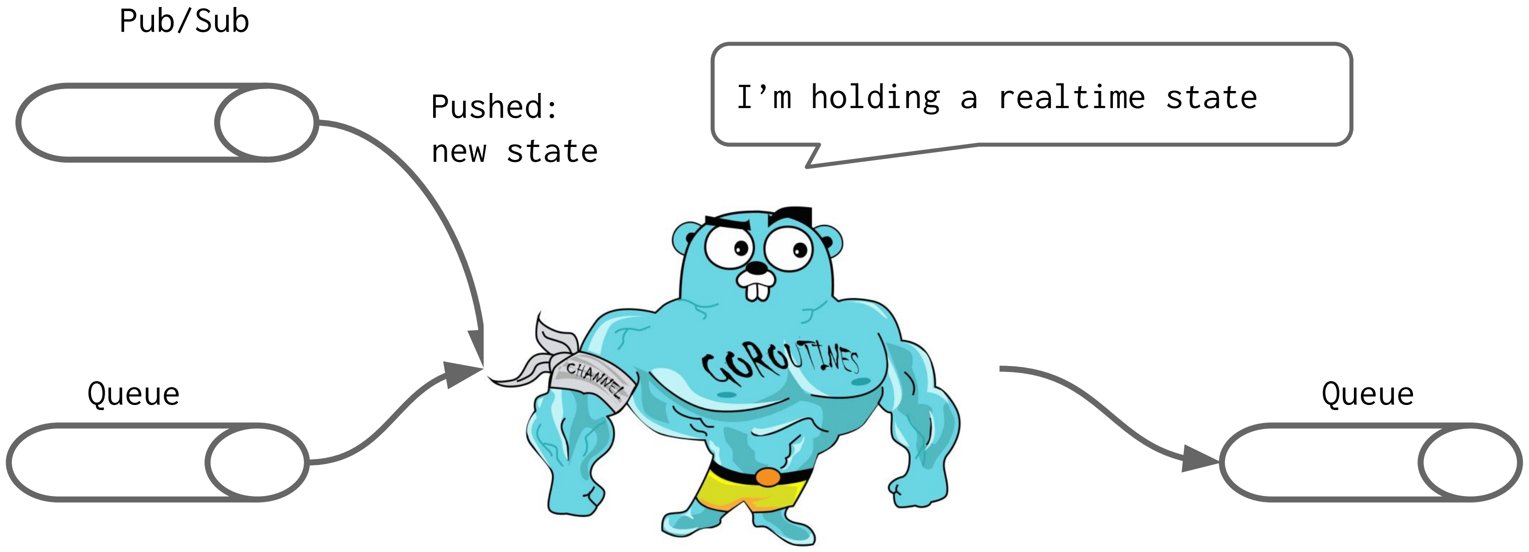
The worker use case



Fan-in : The worker use case

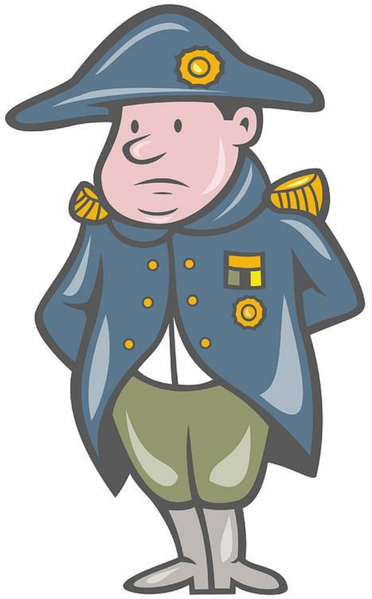
```
“Non real time”
     state
```

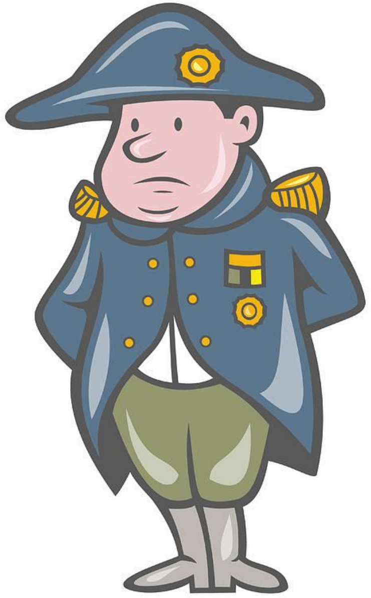
Fan-in : The worker use case



Real world(2) use case

Meet “ลุงประหยัด”

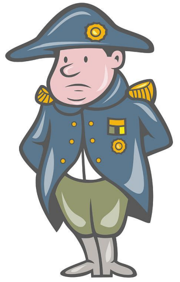


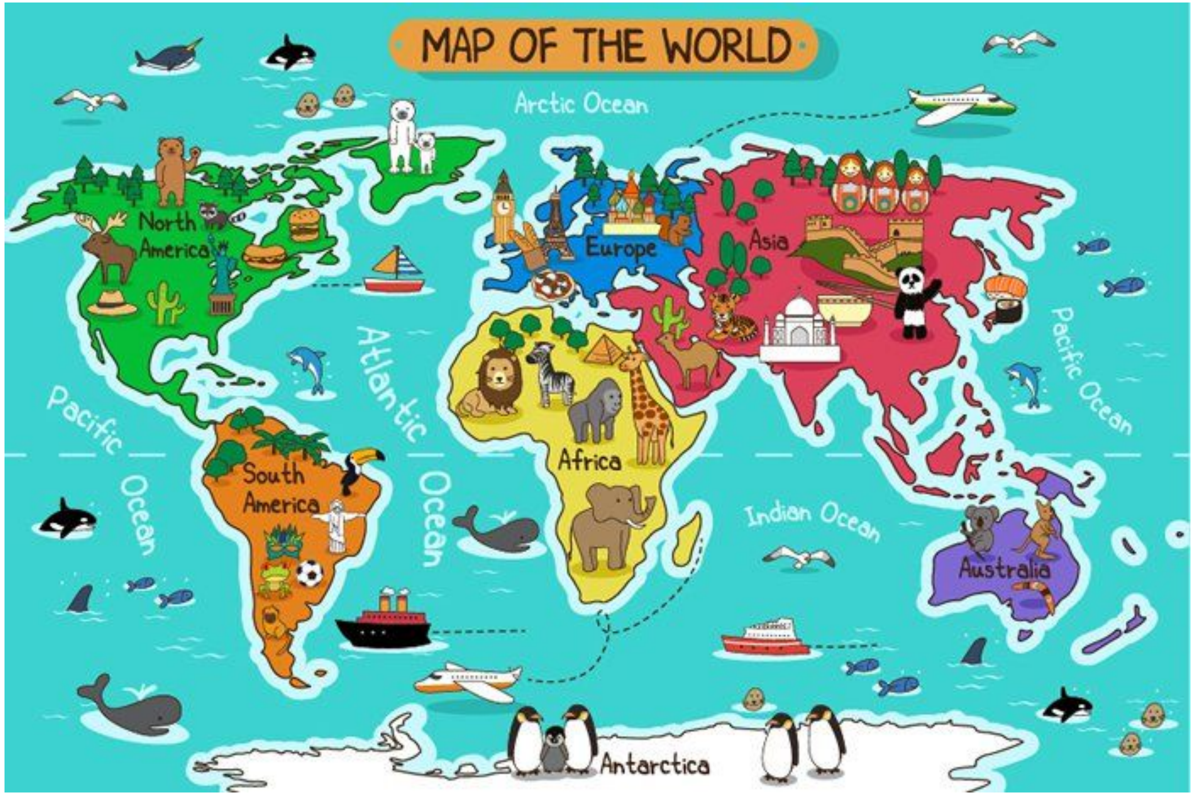


The president of the world(2)

The “ลุงประหยัด” problem

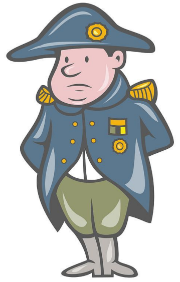
I want everybody in North America to wear **PURPLE** shirt.

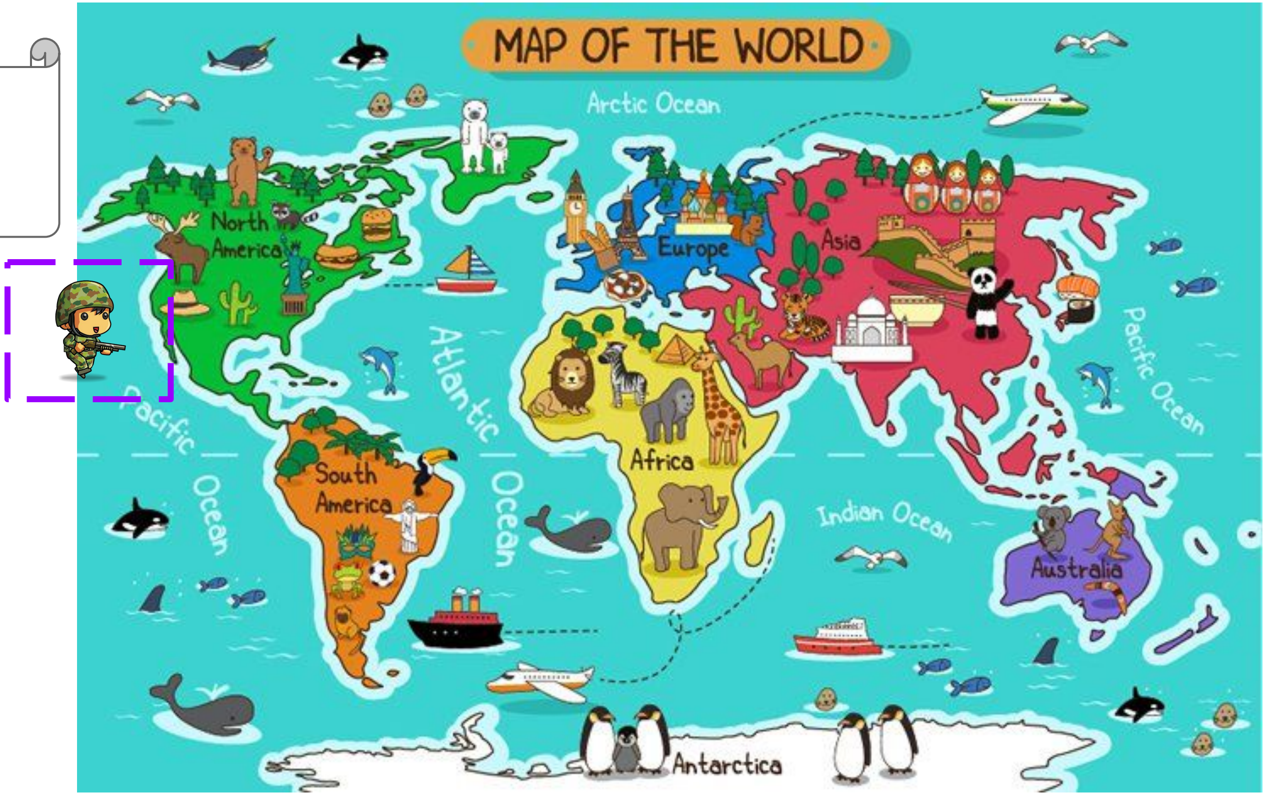




The “ลุงประหยัด” problem

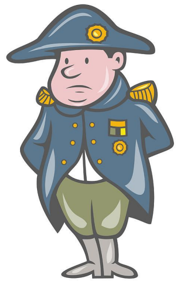
I want everybody in North America to wear **PURPLE** shirt.

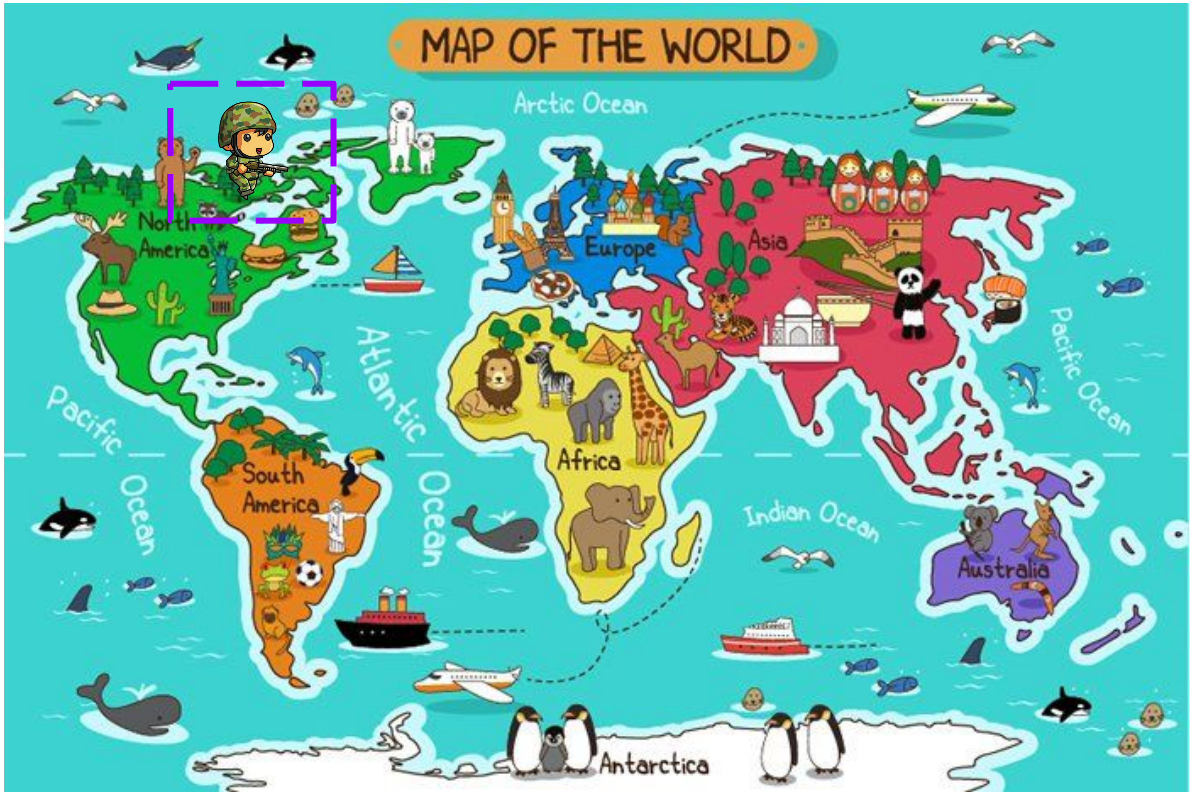




The “ลุงประหยัด” problem

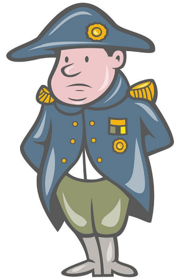
Changed my mind. I want everybody in North America to wear **BLUE** shirt.

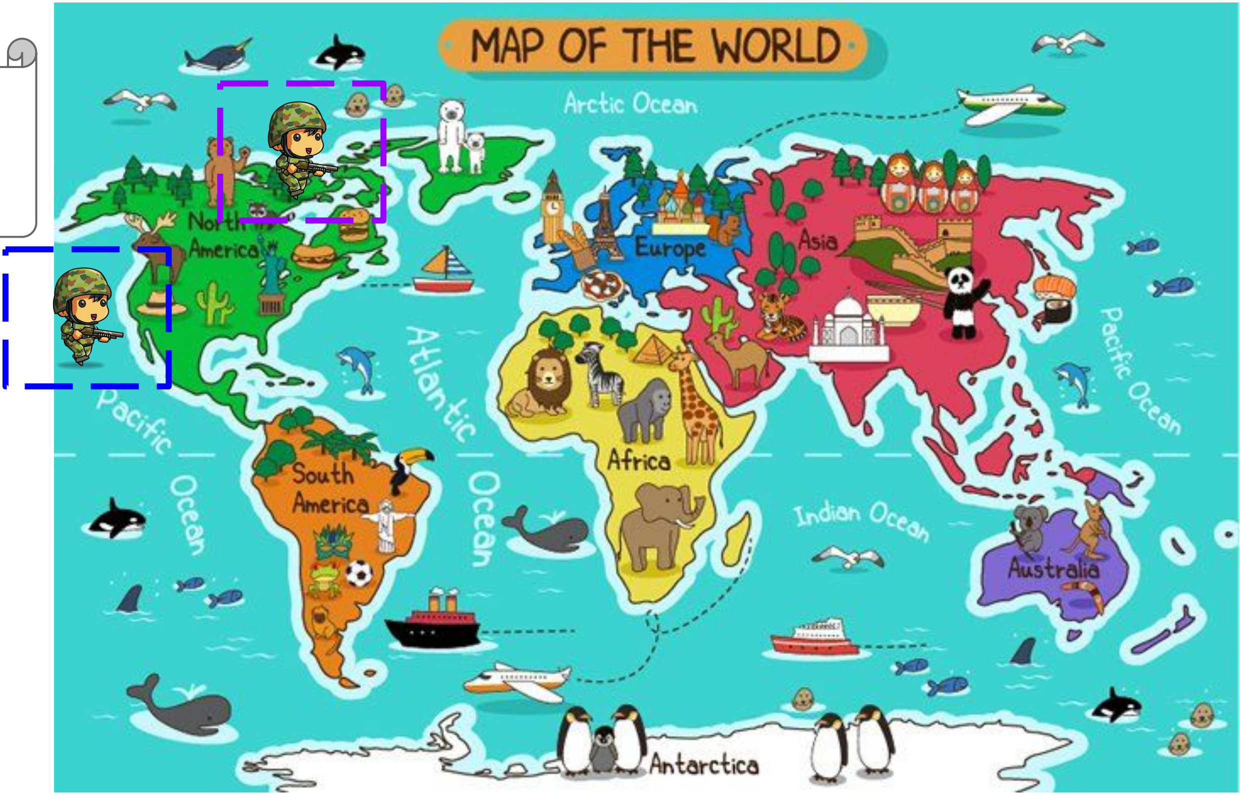




The “ลุงประหยัด” problem

Changed my mind. I want everybody in North America to wear **BLUE** shirt.

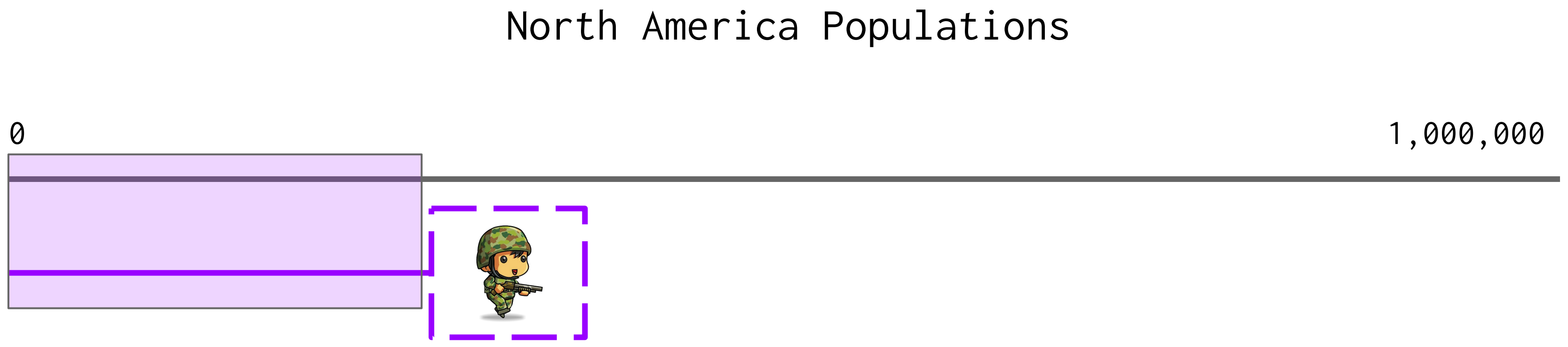




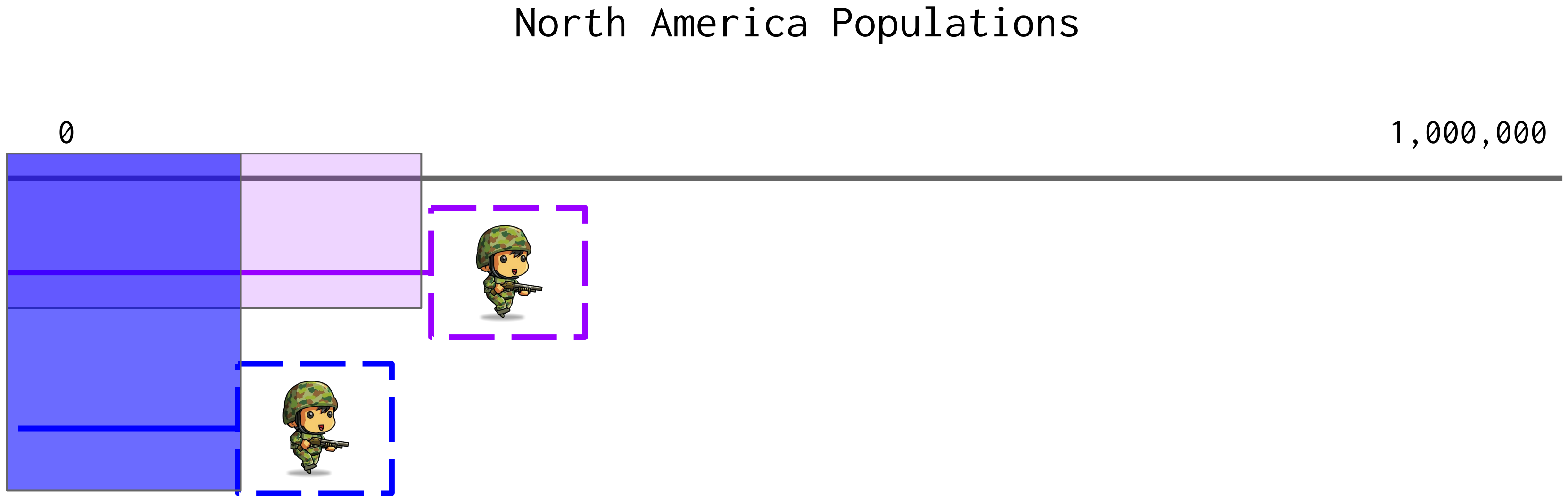
The “ลุงประหยัด” problem

North America Populations
0
1,000,000

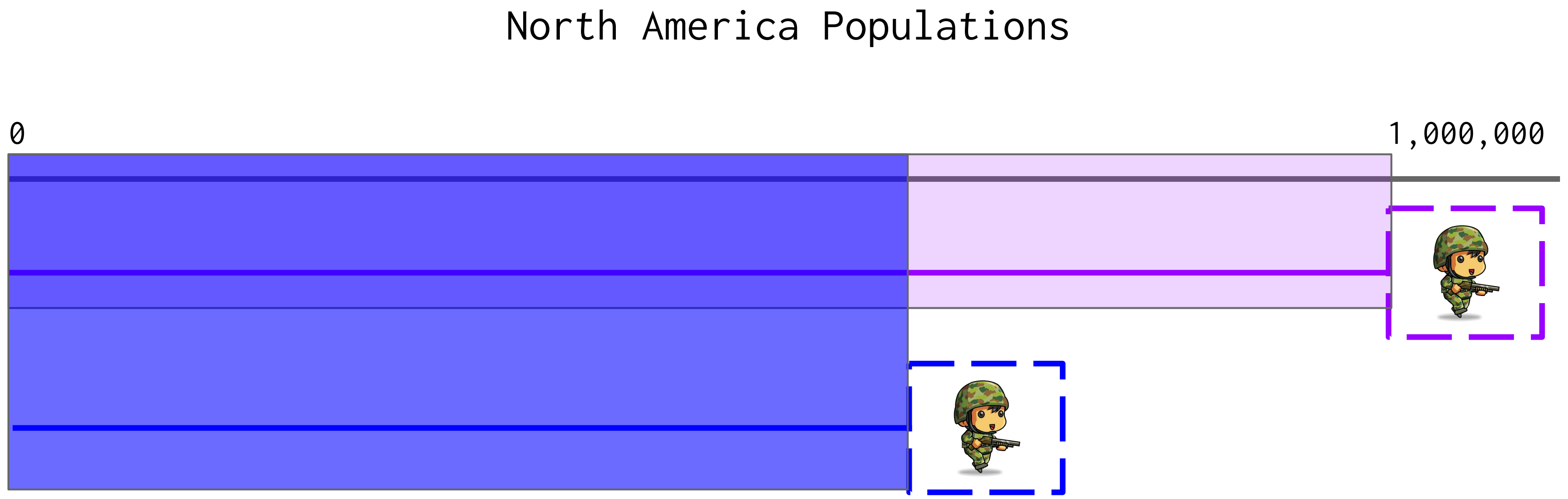
HAPPY PATH ;)



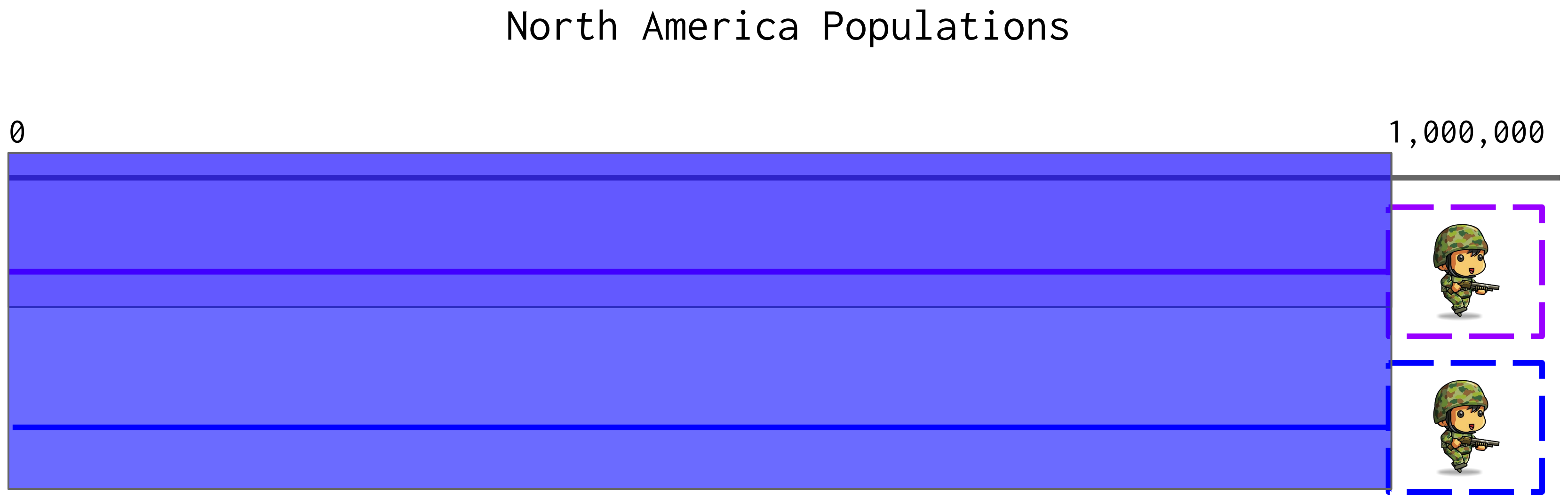
HAPPY PATH ;)



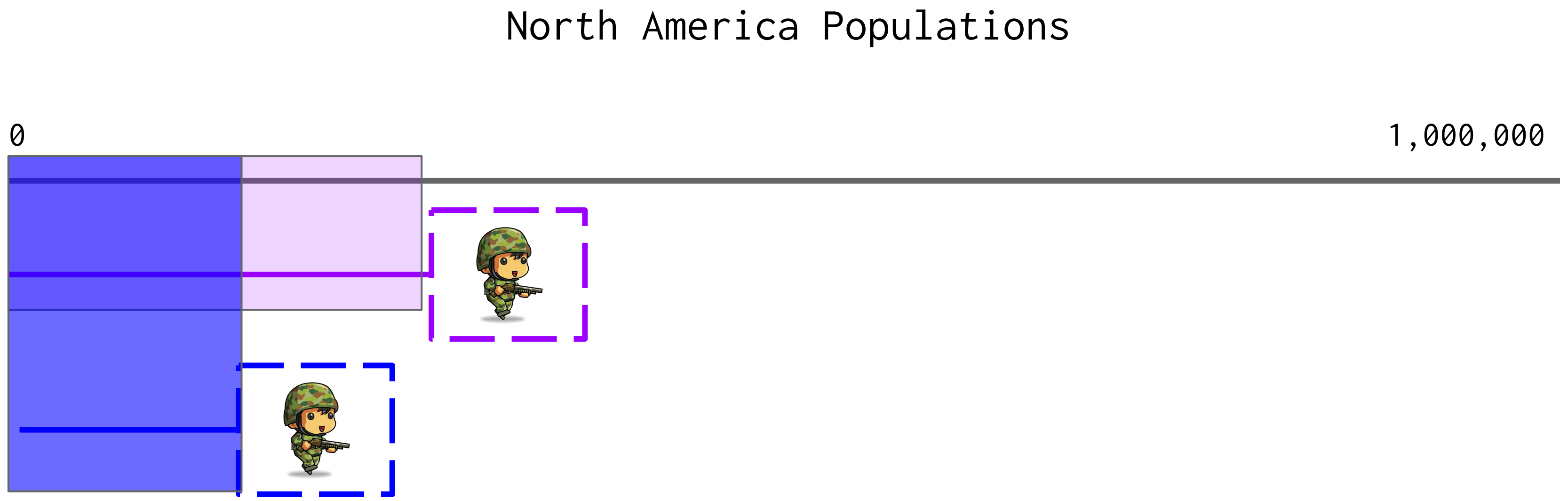
HAPPY PATH ;)



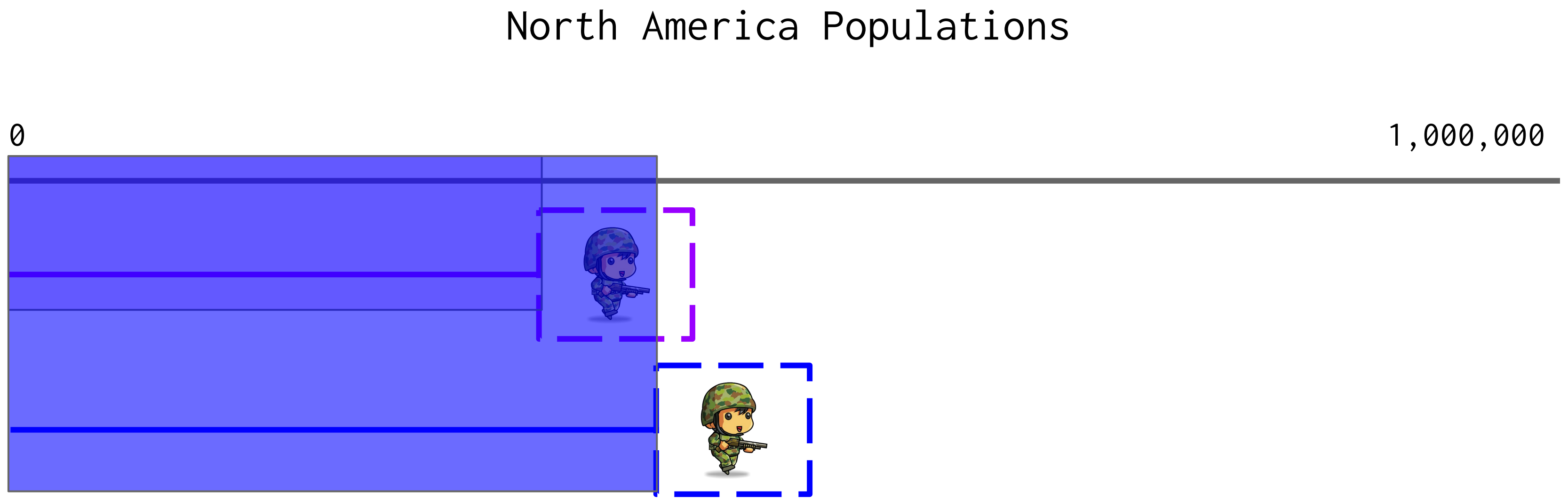
HAPPY PATH ;)



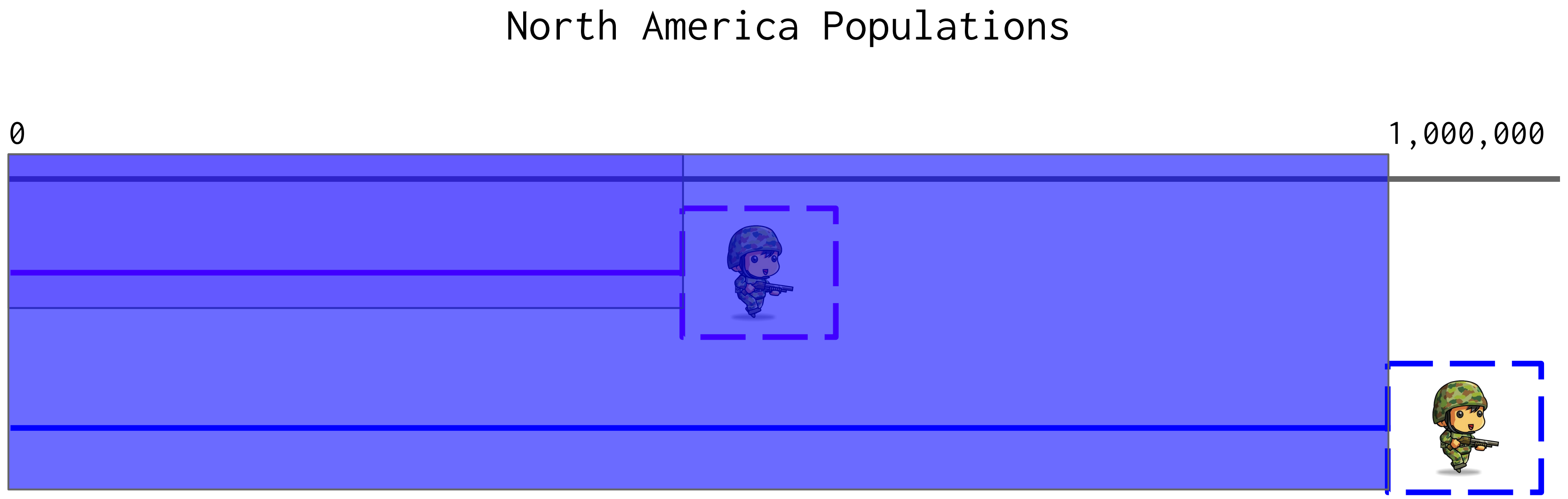
BUG PATH ;(



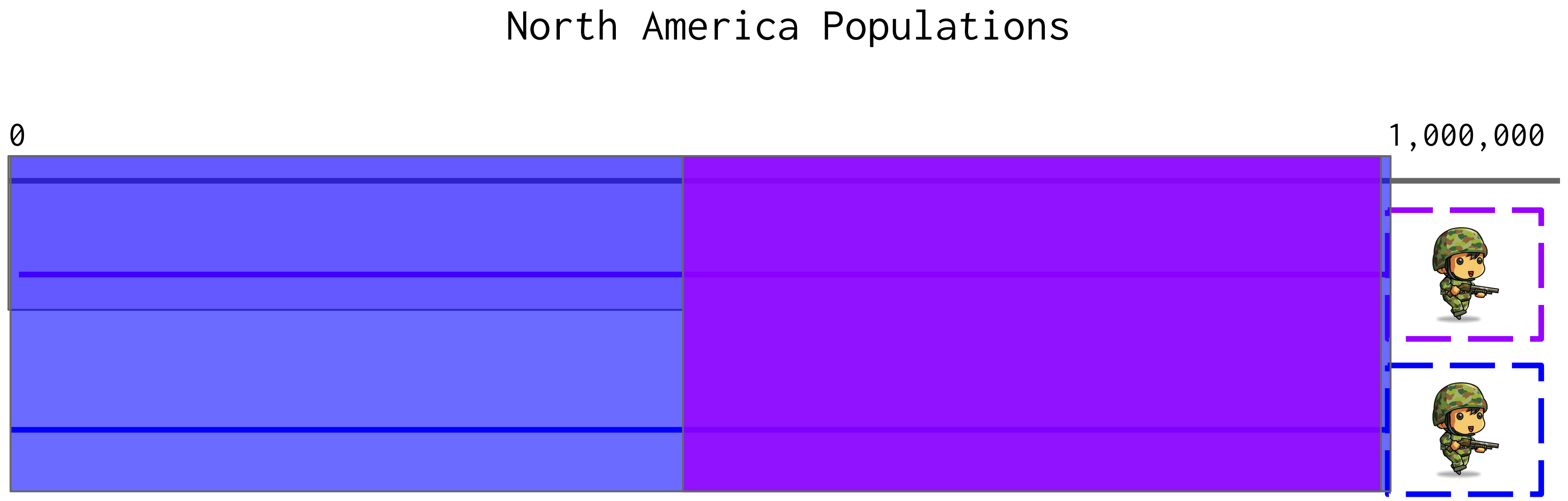
BUG PATH ;(



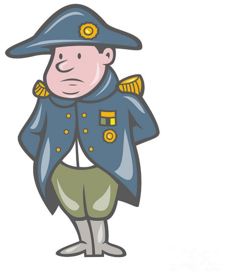
BUG PATH ;(



BUG PATH ;(



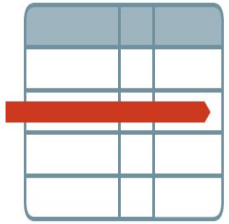
Implementation



= **Job producer**

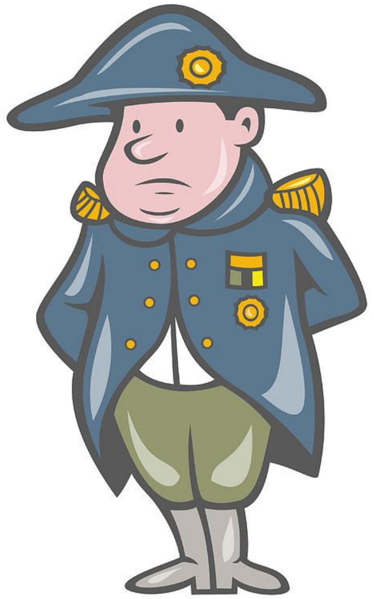
= **Worker**

= **Data** =

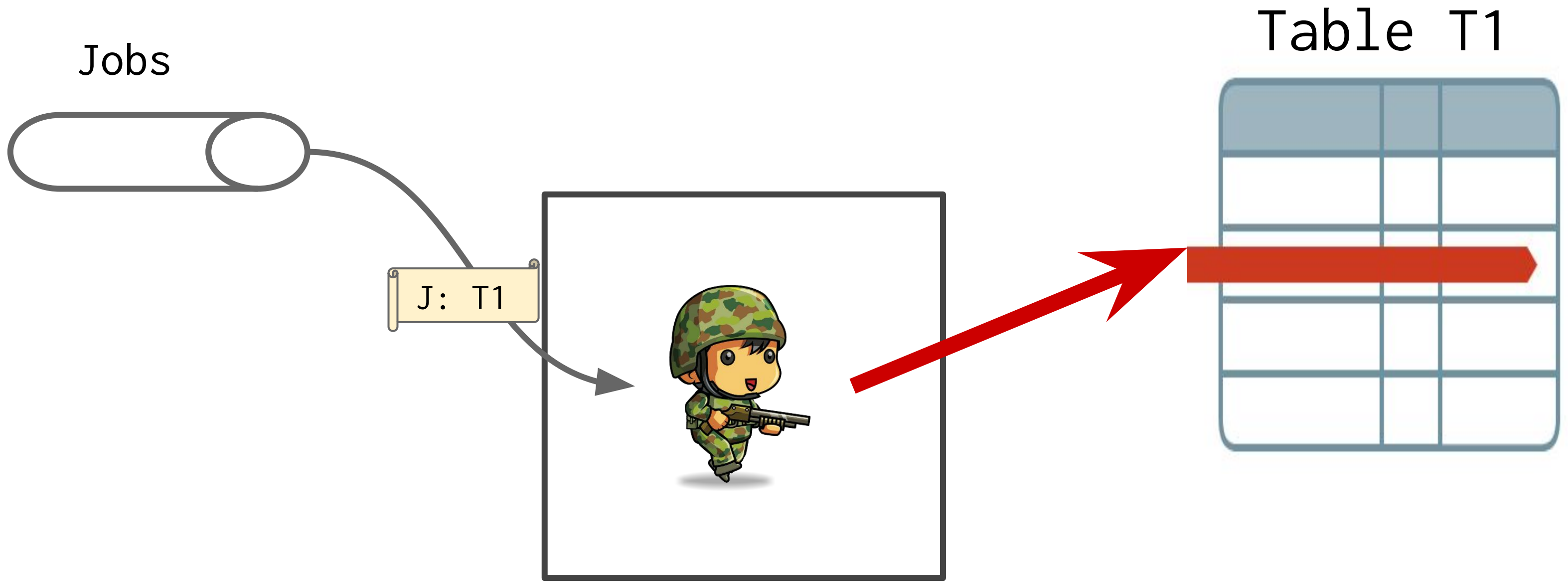


Code v1

“Single node, single worker, single routine”



Overview



“Single node, single worker, single routine”

THE CODE

```
package main

import (...)

func main() {
	jobs //channel attach to rabbitMQ queue
	forever := make(chan bool)
	go func() {
		for j := range jobs {
			dowork(j)
		}
	}()
	<-forever
}
```

```
func dowork(j Job) {
	for i := 0; i < 100; i++ {
		time.Sleep(1 * time.Second)
		fmt.Println(j.Body)
	}
	j.Ack()
}
```

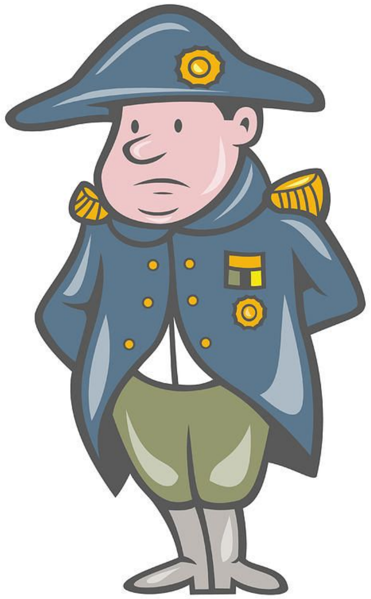
“Single node, single worker, single routine”

Problem: Slow cannot scale

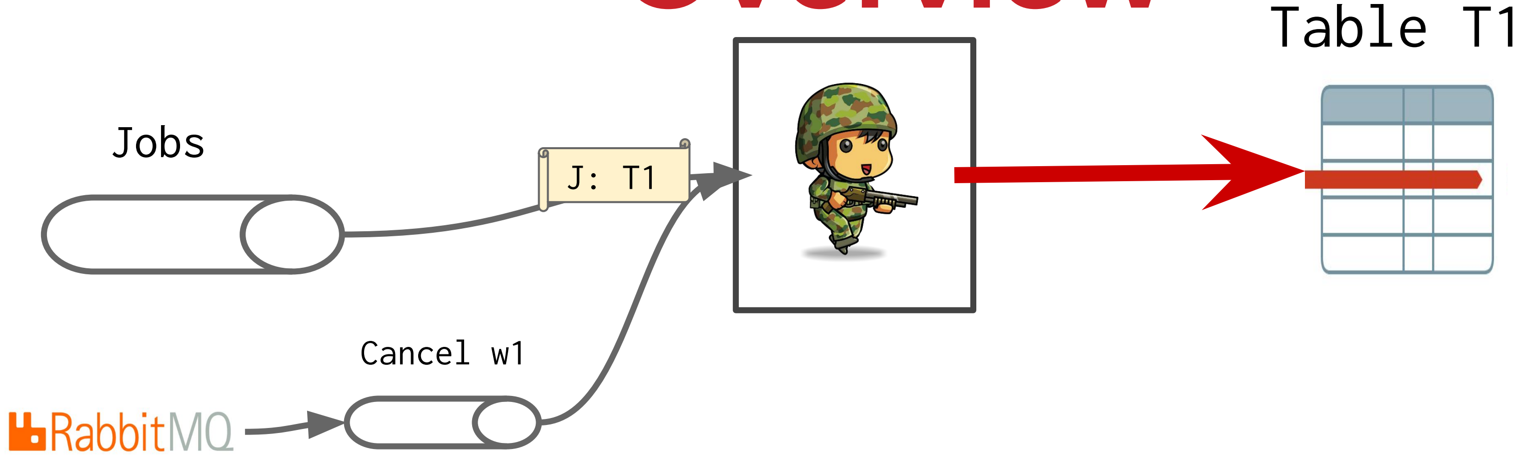
(Scaling introduces the problem)

Code v2

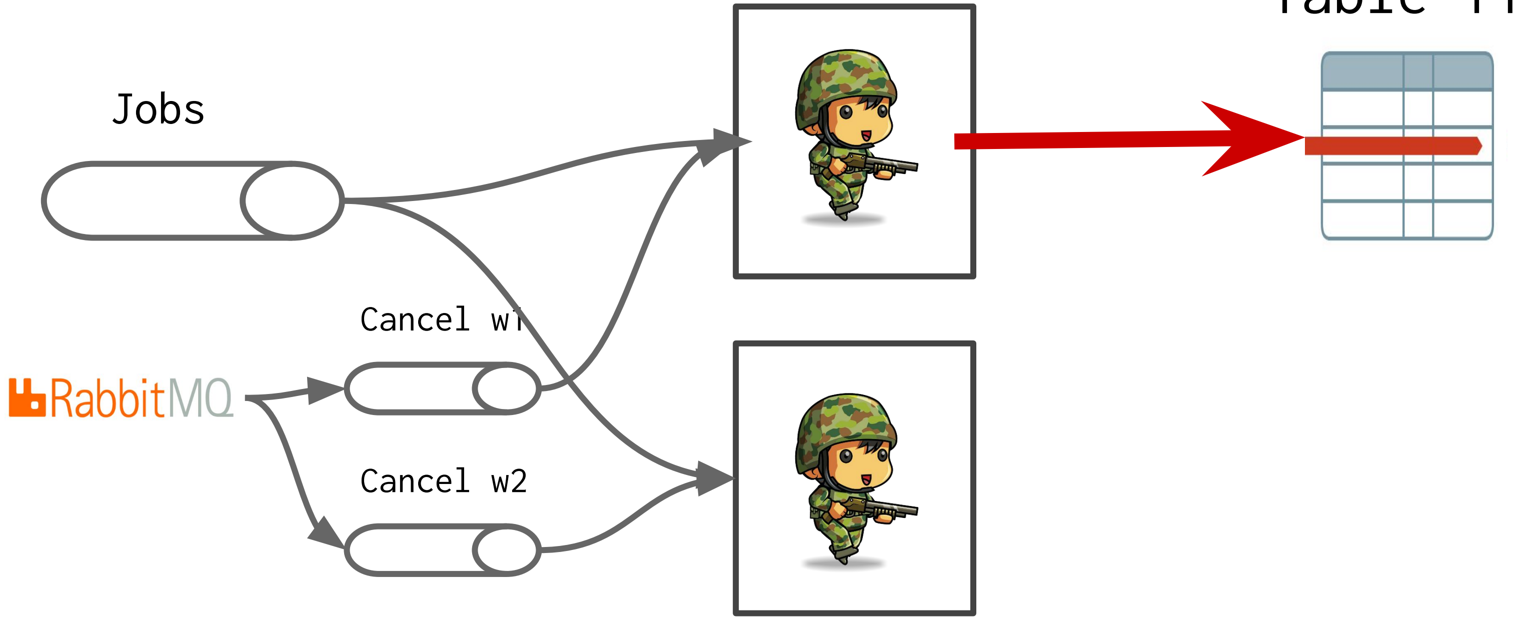
“Multiple nodes, multiple workers, single routine”



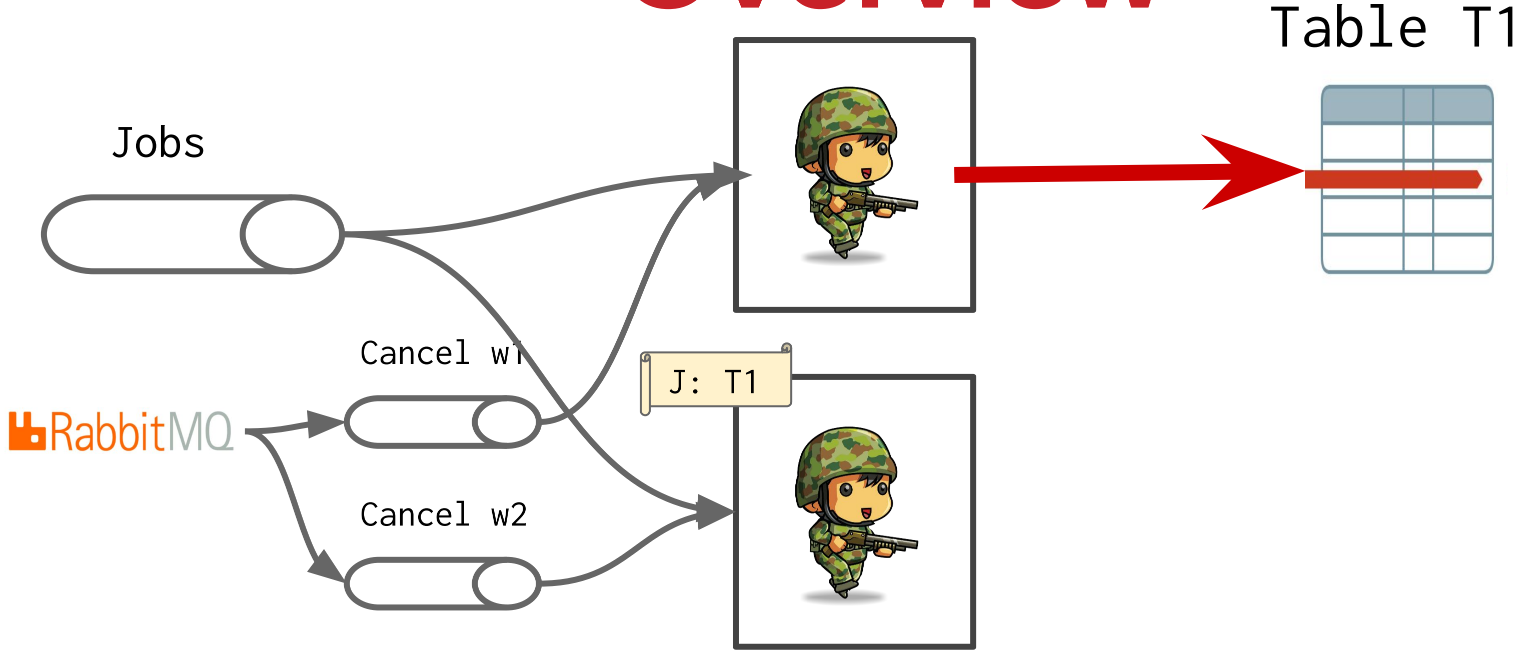
Overview



Overview



Overview



Overview

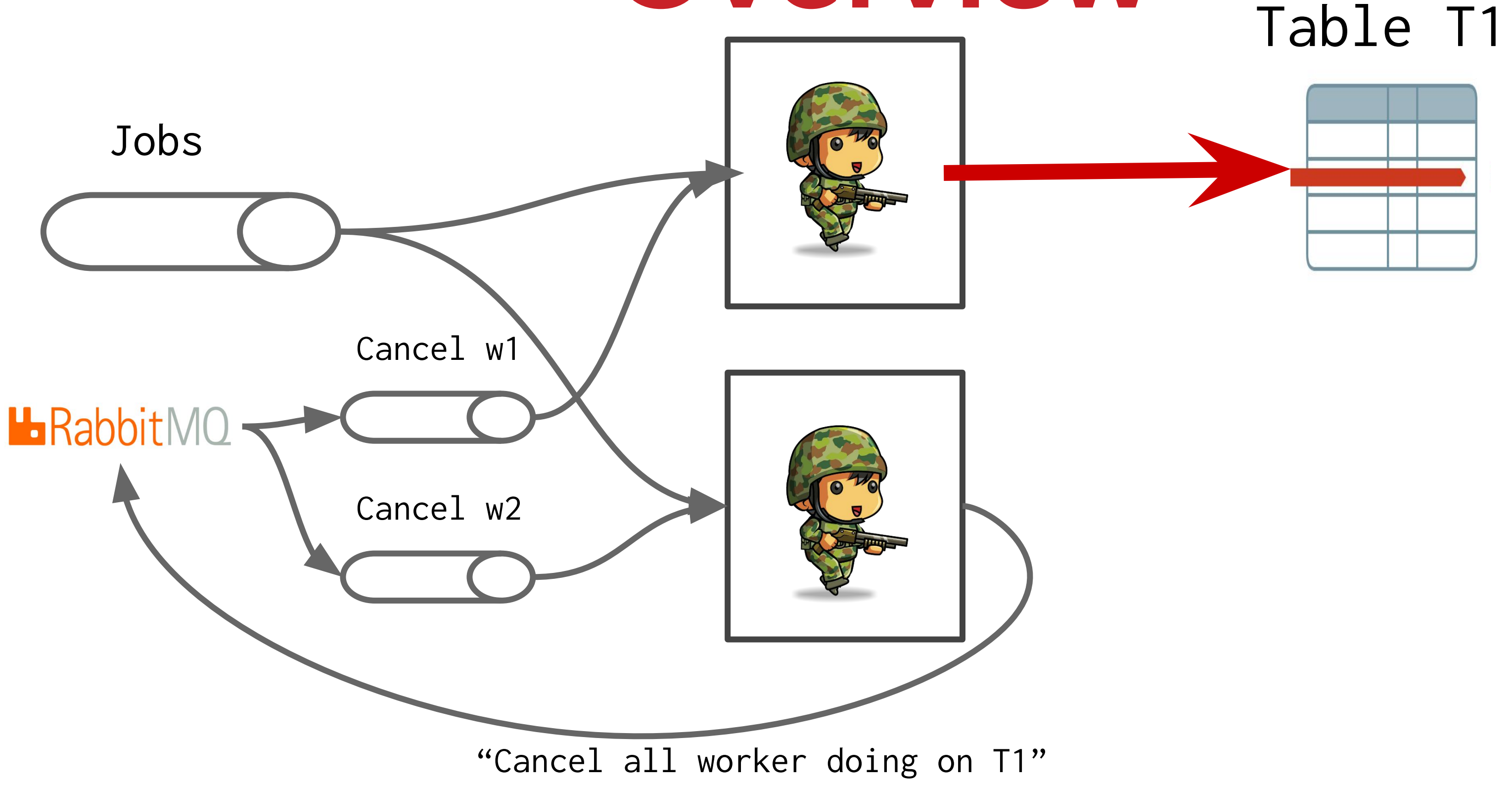


Table T1
Jobs
Cancel w1
Cancel w2
RabbitMQ
“Cancel all worker doing on T1”

Overview

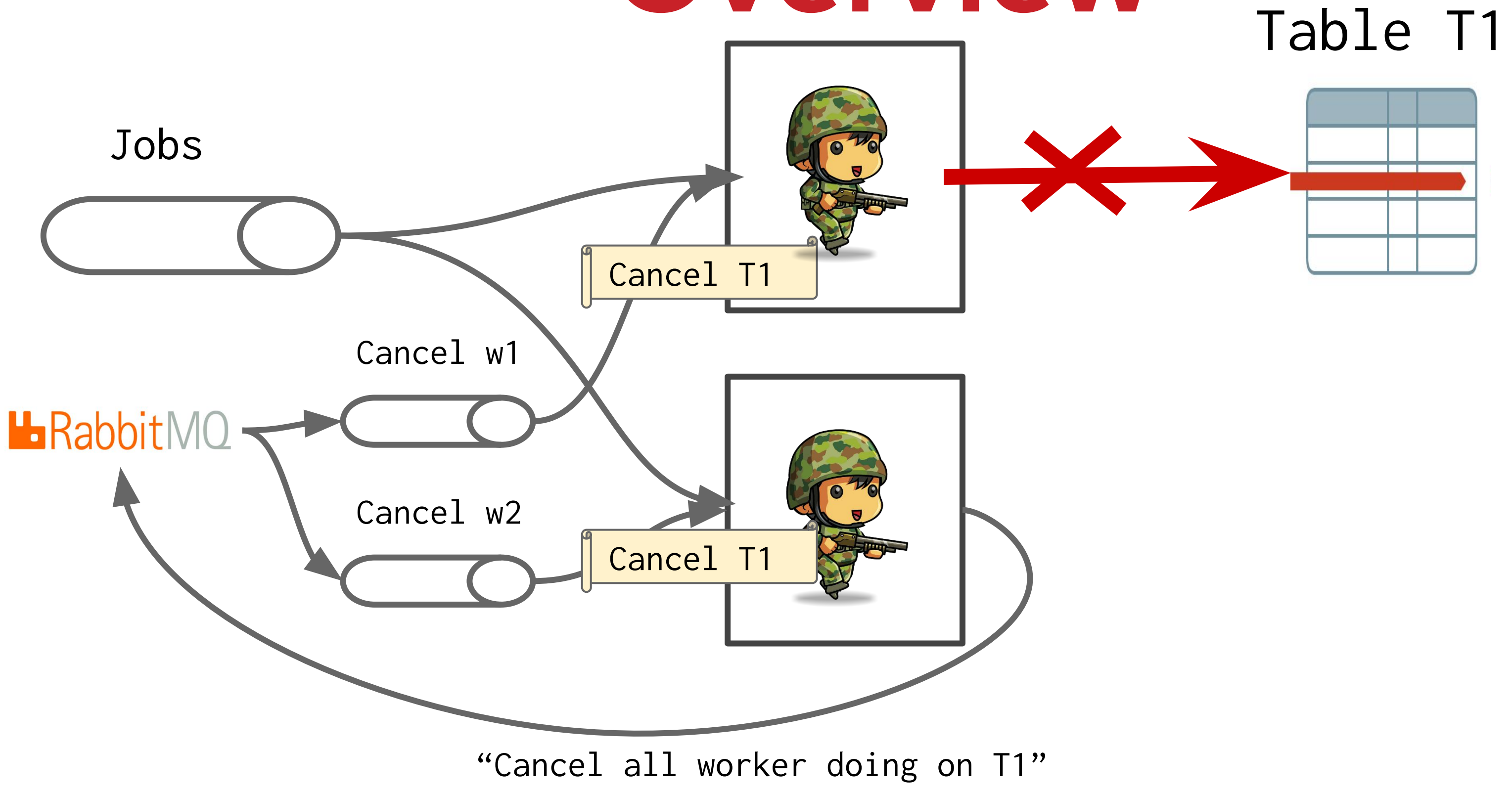
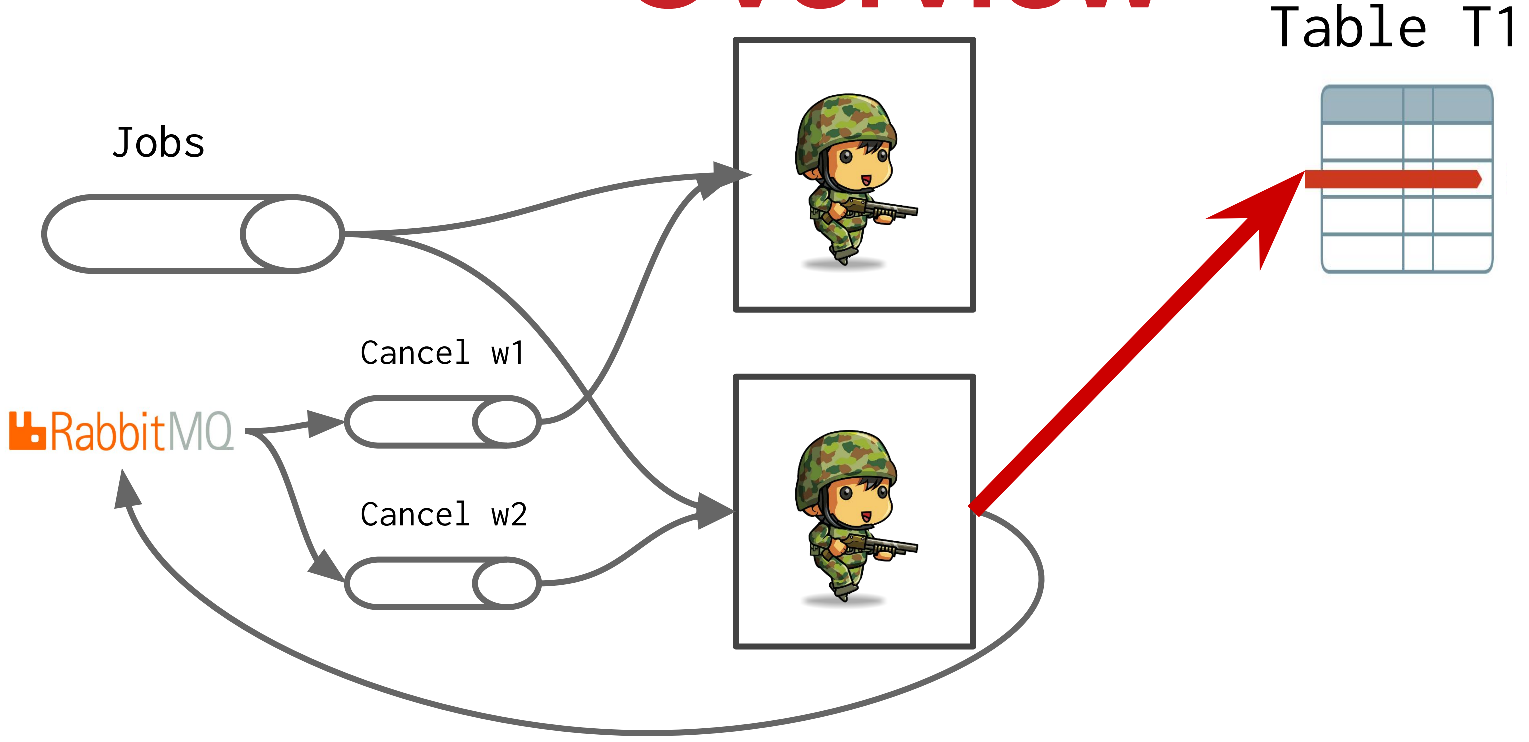


Table T1
Jobs
Cancel T1
Cancel w1
RabbitMQ
Cancel w2
Cancel T1
“Cancel all worker doing on T1”

Overview



“Cancel all worker doing on T1”

THE CODE

```
package main
import (...)
func main() {
	jobs //channel attach to rabbitMQ queue
	cancelJobs //channel attach to rabbitMQ pub/sub
	cancelBroadcaster //pub to rabbitMQ pub/sub
	stopChan := make(chan bool)
	forever := make(chan bool)
	sem := make(chan bool, 1)
	currentCommand := ""
	go func() {
		for {
			select {
			case j  := <-jobs:
				cmd := j.Body; currentCommand = cmd
				cancelBroadcaster.Publish(
					CancelJob{cmd,consumerID}
				)
				sem <- true
				go dowork(j, stopChan, sem)
			case cj := <-cancelJobs:
				if cj.ConsumerID != consumerID &&
					cj.Command == currentCommand {
					// Stop (another worker is going to do)
					close(stopChan)
					stopChan = make(chan bool)
				}
			}
```

```
func dowork(j Job, stopChan chan bool, sem
chan bool) {
	counter := 0
dowork:
	for {
		select {
		case _, more := <-stopChan:
			if !more {
				break dowork
			}
		default:
			if counter < 100 {
				counter++
				time.Sleep(..)
				fmt.Println(j.Body)
			}
		}
	}
	j.Ack()
	<-sem
}
```

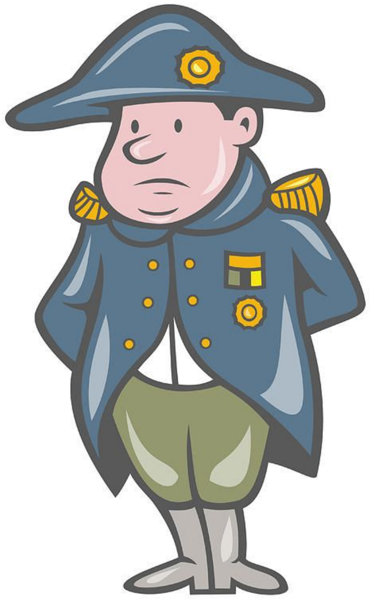
`“Multiple nodes, multiple workers, single routine”`

Problem: Performance

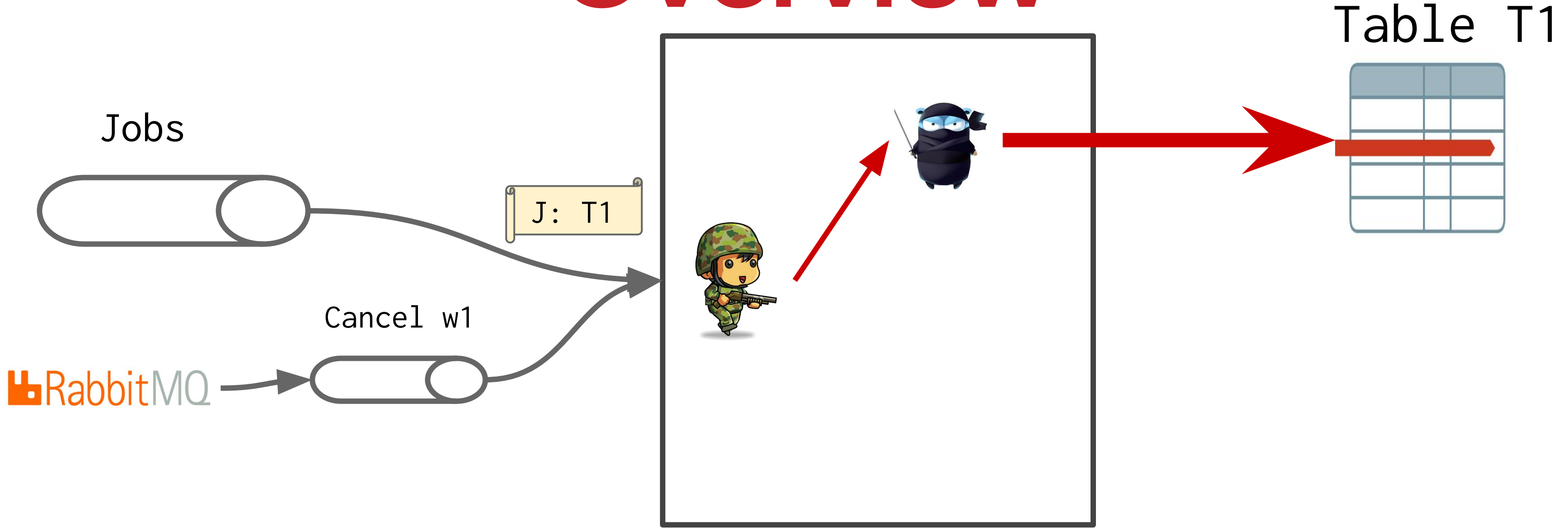
(Does not yet utilize goroutine)

Code v3

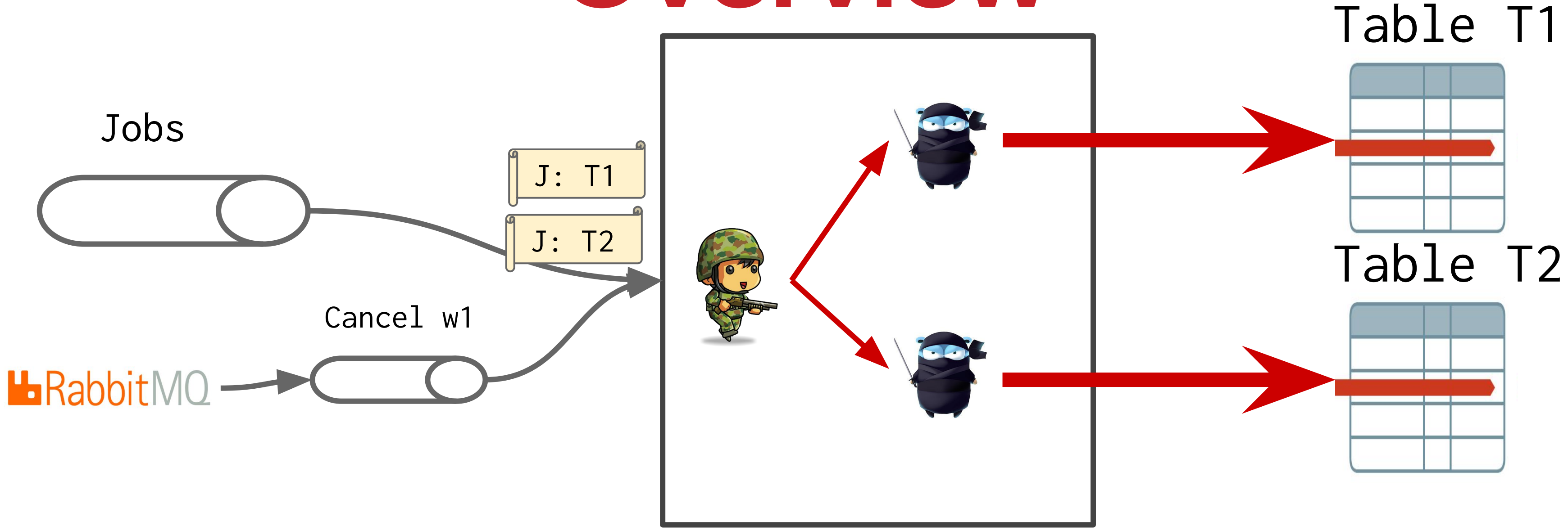
“Multiple nodes, multiple workers, multiple routines”



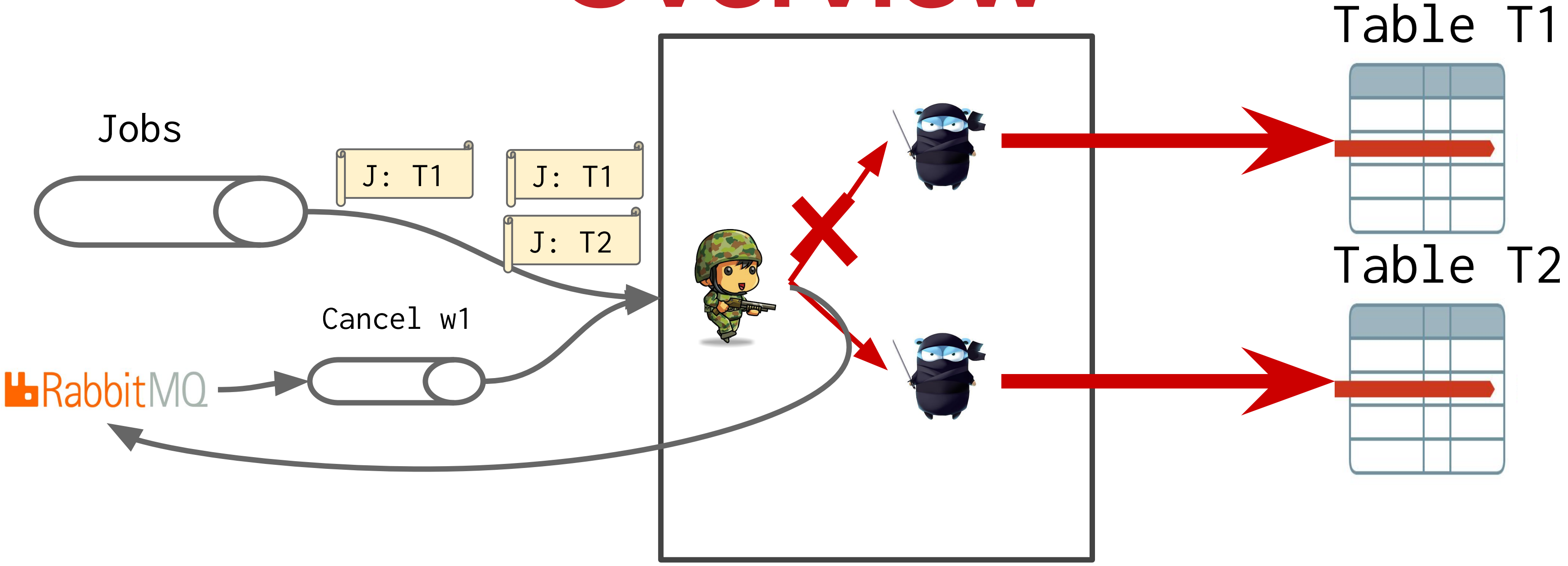
Overview



Overview

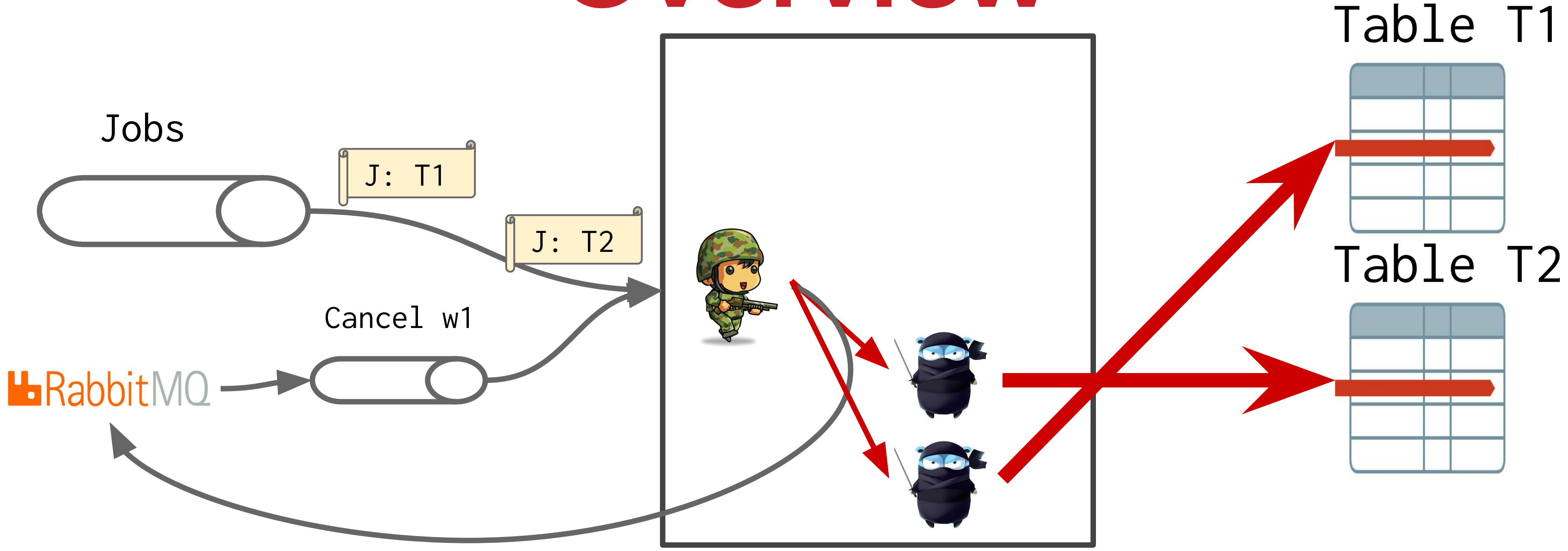


Overview



“Cancel all worker doing on T1”

Overview



“Cancel all worker doing on T1”

THE CODE

```go
package main

import (...)

func main() {
      jobs //channel attach to rabbitMQ queue
      cancelJobs//channel attach to rabbit pub/sub
      cancelBroadCaster //pub to rabbitMQ pub/sub
      forever := make(chan bool)
      runningRoutines := make(map[string]chan bool)
      doneChan := make(chan string)
      go func() {
            for {
                  select {
                  case cmd := <-doneChan:
                  case j  := <-jobs:
                  case cj := <-cancelJobs:
                  }
            }
      }()
      <-forever
}
```

```go
case cmd := <-doneChan:
      close(runningRoutines[cmd])
      delete(runningRoutines, cmd)

case j := <-jobs:
      cmd := j.Body
      cancelBroadCaster.Publish(CancelJob{cmd, consumerID})
      if stopChan,exist := runningRoutines[cmd]; exist {
            // Cancelled same job in same machine
            close(stopChan)
      }
      runningRoutines[cmd] = make(chan bool)
      go dowork(j, runningRoutines[cmd], doneChan)

case cj := <-cancelJobs:
      if cj.ConsumerID != consumerID {
            if stopChan,e := runningRoutines[cj.Command]; e {
            // Going to Cancel routine
            // (New job running on diff machine)
                  close(stopChan)
                  delete(runningRoutines, cj.Command)
            }
      }
```

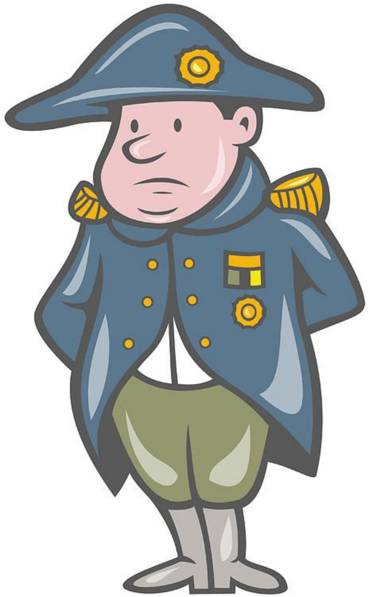
THE CODE

```
func dowork(j Job, stopChan chan bool, doneChan chan string) {
	counter := 0
dowork:
	for {
		select {
		case _, more := <-stopChan:
			if !more {
				break dowork
			}
		default:
			if counter < 100 {
				counter++
				time.Sleep(1 * time.Second)
				fmt.Println(j.Body)
			} else {
				cmd := j.Body
				doneChan <- cmd
				break dowork
			}
		}
	}
	j.Ack()
}
```

Code:

github.com/naponmeka/synchronization_go_workers

Thank you ;)



One more thing...