Concurrency Patterns in Go

CS3211 Parallel and Concurrent Programming

Outline

- Patterns
    - Confinement
    - For-select loop
    - Preventing goroutine leaks
    - Error handling
    - Pipeline
    - Fan-out, fan-in
- A load balancing example

Patterns

- Separation of concerns
  - Data chunks (confinement)
  - Error handling
  - Data processing (pipeline)

Confinement

- Achieve safe operation
  - Synchronization primitives for sharing memory (e.g., sync.Mutex)
  - Synchronization via communicating (e.g., channels)
- Safe concurrency with good performance
  - Immutable data
  - Data protected by confinement

Achieving confinement

- Ad-hoc confinement
  - By convention, data is modified only from one goroutine, even though it is accessible from multiple goroutines
  - Needs some static analysis to ensure safety
- Lexical confinement
  - Restrict the access to shared locations

Lexical confinement (1)

- Lines 4 and 14: expose only the reading/writing handle of the channel

```go
4  chanOwner := func() <-chan int {
5      results := make(chan int, 5)
6      go func() {
7          defer close(results)
8          for i := 0; i <= 5; i++ {
9              results <- i
10         }
11     }()
12     return results
13 }
14 consumer := func(results <-chan int) {
15     for result := range results {
16         fmt.Printf("Received: %d\n", result)
17     }
18     fmt.Println("Done receiving!")
19 }
20 results := chanOwner()
21 consumer(results)
22 }
```

Lexical confinement (2)

- Lines 35-36:Expose only a slice of the array

```go
24 printData := func(wg *sync.WaitGroup, data []byte) {
25     defer wg.Done()
26     var buff bytes.Buffer
27     for _, b := range data {
28         fmt.Fprintf(&buff, "%c", b)
29     }
30     fmt.Println(buff.String())
31 }
32 var wg sync.WaitGroup
33 wg.Add(2)
34 data := []byte("golang")
35 go printData(&wg, data[:3])
36 go printData(&wg, data[3:])
37 wg.Wait()
```

The for-select loop

- Context
  - Sending iteration variables out on a channel
  - Looping and waiting to be stopped

```
for { // Either loop infinitely or range over something
    select {
    // Do some work with channels
    }
}
```

From tutorial 5

- Loop
- Keeps the `select` statement as short as possible
- Do work while done channel is not closed

```go
func consumer(done chan struct{}, q chan int, sumCh chan int) {
    sum := 0
    for {
        select {
        case num := <-q:
            sum += num
        case <-done:
            sumCh <- sum
            return
        }
    }
}
```

Preventing goroutines from leaking

- Goroutine *do* cost resources!
- Ensure termination of your goroutines
  - When it has completed its work
  - When it cannot continue its work due to an unrecoverable error
  - When it’s told to stop working
- Convention: if a goroutine is responsible for creating a goroutine, it is also responsible for ensuring it can stop the goroutine

Leaking goroutines

- Line 16: goroutines will accumulate in memory
- Leaking because
  - a `nil` channel was passed and
  - Line 9: forever blocks due to semantics of reading from a `nil` channel

```
4   doWork := func(strings <-chan string) <-chan interface{} {
5       completed := make(chan interface{})
6       go func() {
7           defer fmt.Println("doWork exited.")
8           defer close(completed)
9           for s := range strings {
10              // Do something interesting
11              fmt.Println(s)
12          }
13      }()
14      return completed
15  }
16  doWork(nil)
17  // Perhaps more work is done here
18  fmt.Println("Done.")
```

Stopping reader goroutines (1)

- Line 25: done channel passed to the `doWork` function
- Line 26: another goroutine will cancel the goroutine spawned from `doWork` if more than one second passes
- Line 32: join the goroutine spawned from `doWork` with the `main` goroutine

```go
24    done := make(chan interface{})
25    terminated := doWork(done, nil)
26    go func() {
27        // Cancel the operation after 1 second.
28        time.Sleep(1 * time.Second)
29        fmt.Println("Canceling doWork goroutine...")
30        close(done)
31    }()
32    <-terminated
33    fmt.Println("Done.")
```

Stopping reader goroutines (2)

- Lines 12-19: for-select pattern in use

```
4  doWork := func(
5      done <-chan interface{},
6      strings <-chan string,
7  ) <-chan interface{} {
8      terminated := make(chan interface{})
9      go func() {
10         defer fmt.Println("doWork exited.")
11         defer close(terminated)
12         for {
13             select {
14             case s := <-strings:
15                 // Do something interesting
16                 fmt.Println(s)
17             case <-done:
18                 return
19             }
20         }
21     }()
22     return terminated
23 }
```

Stopping writer goroutines (2)

```go
newRandStream := func(done <-chan interface{}) <-chan int {
    randStream := make(chan int)
    go func() {
        defer fmt.Println("newRandStream closure exited.")
        defer close(randStream)
        for {
            select {
            case randStream <- rand.Int():
            case <-done:
                return
            }
        }
    }()
    return randStream
}
```

```go
    done := make(chan interface{})
    randStream := newRandStream(done)
    fmt.Println("3 random ints:")
    for i := 1; i <= 3; i++ {
        fmt.Printf("%d: %d\n", i, <-randStream)
    }
    close(done)
    // Simulate ongoing work
    time.Sleep(1 * time.Second)
}
```

Error handling

- Goal: gracefully handle erroneous states
- Responsibility for handling errors
    - A goroutine to maintain complete information about the state of the program
    - All goroutines send their errors to the state-goroutine that can make an informed decision about what to do
    - Couple the potential result with the potential error
        - Errors should be tightly coupled with your result type, and passed along through the same lines of communication

Error handling example

```go
type Result struct {
	Error error
	Response *http.Response
}
```

```go
checkStatus := func(done <-chan interface{}, urls ...string) <-chan Result {
	results := make(chan Result)
	go func() {
		defer close(results)
		for _, url := range urls {
			var result Result
			resp, err := http.Get(url)
			result = Result{Error: err, Response: resp}
			select {
			case <-done:
				return
			case results <- result:
			}
		}
	}()
	return results
}
```

```go
done := make(chan interface{})
defer close(done)
urls := []string{"https://www.google.com", "https://badhost"}
for result := range checkStatus(done, urls...) {
	if result.Error != nil {
		fmt.Printf("error: %v", result.Error)
		continue
	}
	fmt.Printf("Response: %v\n", result.Response.Status)
}
```

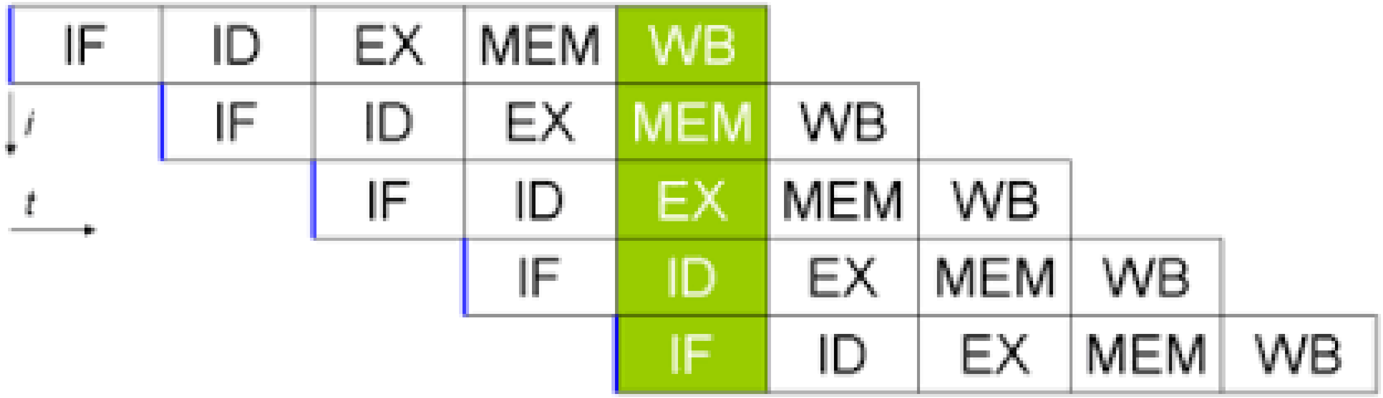
Error handling example

```go
type Result struct {
    Error error
    Response *http.Response
}
```

```go
checkStatus := func(done <-chan interface{}, urls ...string) <-chan Result {
    results := make(chan Result)
    go func() {
        defer close(results)
        for _, url := range urls {
            var result Result
            resp, err := http.Get(url)
            result = Result{Error: err, Respon
            select {
            case <-done:
                return
            case results <- result:
            }
        }
    }()
    return results
}
```

```go
done := make(chan interface{})
defer close(done)
errCount := 0
urls := []string{"a", "https://www.google.com", "b", "c", "d"}
for result := range checkStatus(done, urls...) {
    if result.Error != nil {
        fmt.Printf("error: %v\n", result.Error)
        errCount++
        if errCount >= 3 {
            fmt.Println("Too many errors, breaking!")
            break
        }
        continue
    }
    fmt.Printf("Response: %v\n", result.Response.Status)
}
```

Pipeline



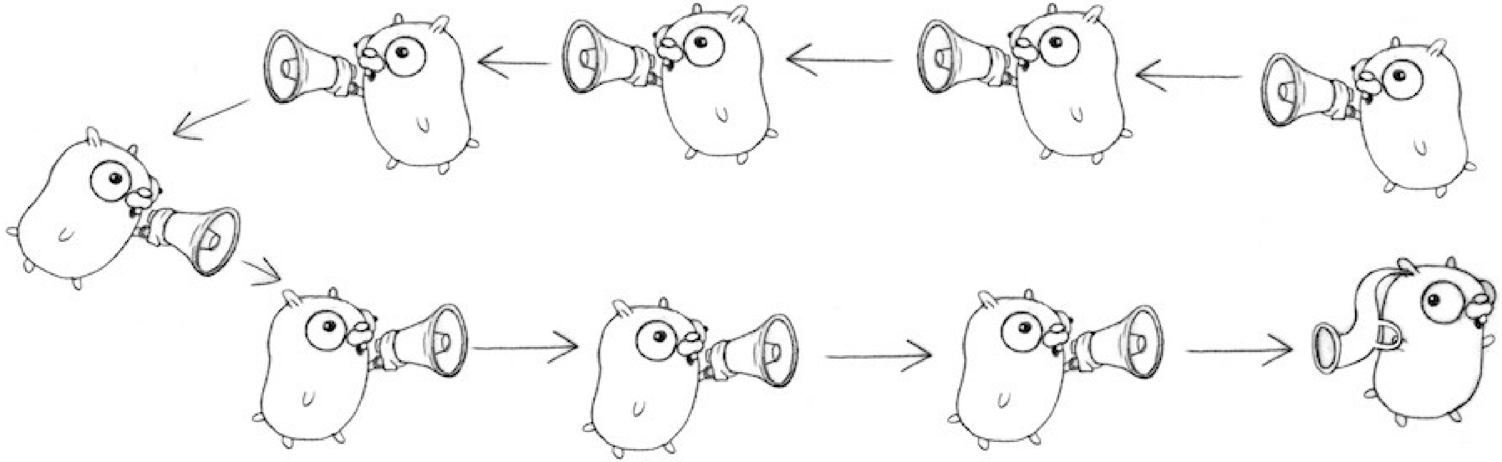
- Multiple types
  - Instruction pipelines
  - Graphics pipelines
  - Software pipelines
- A set of data processing elements connected in series, where the output of one element is the input of the next one
  - Stages
  - Connect the stages

Pipelines in Go

- A series of stages connected by channels
  - Each stage is a group of goroutines running the same function
- In each stage, the goroutines
  - receive values from upstream via inbound channels
  - perform some function on that data, usually producing new values
  - send values downstream via outbound channels
- Each stage has any number of inbound and outbound channels, except the first and last stages
  - The first stage: source or producer
  - The last stage: the sink or consumer

Pipeline pattern in concurrent programming

- Separate the concerns of each stage
  - Modify stages independently of one another,
  - Mix and match how stages are combined independent of modifying the stages
  - Process each stage concurrent to upstream or downstream stages
  - Fan-out, or rate-limit portions of your pipeline



Pipelines design

- For efficiency
  - Designer should divide the work and resources among the stages such that they all take the same time to complete their tasks
  - Fan-out to decrease the processing time for a stage if that is the bottleneck
  - Use of I/O and multiple CPUs for processing streams of data
- Not obviously faster than a task pool
  - Tweaking is needed to make the pipeline more efficient than a task pool
  - Pipeline is better if there is a cap on a specific resource that is needed by all tasks in the task pool (at different times)
    - For example, reading or writing to a restricted network link

How it works?

Iteration	Generator	Multiply	Add	Multiply	Value
0	1				
0		1			
0	2		2		
0		2		3	
1	3		4		6
close(done)	(closed)	3		5	
		(closed)	6		
			(closed)	7	
				(closed)	
					(exit range)

```go
done := make(chan interface{})
defer close(done)
intStream := generator(done, 1, 2, 3, 4)
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
for v := range pipeline {
    fmt.Println(v)
}
```

Pipeline Example

```go
generator := func(done <-chan interface{},
    integers ...int
) <-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                return
            case intStream <- i:
            }
        }
    }()
    return intStream
}
```

```go
multiply := func(
    done <-chan interface{},
    intStream <-chan int,
    multiplier int,
) <-chan int {
    multipliedStream := make(chan int)
    go func() {
        defer close(multipliedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case multipliedStream <- i*multiplier:
            }
        }
    }()
    return multipliedStream
}
```

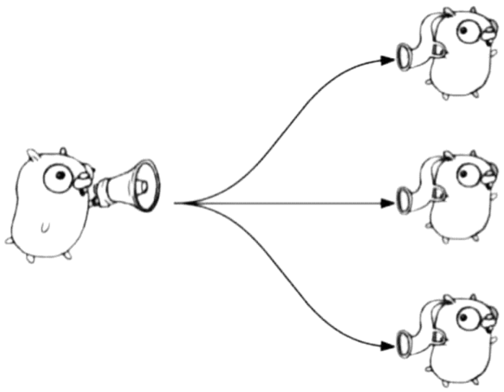
Pipeline Example

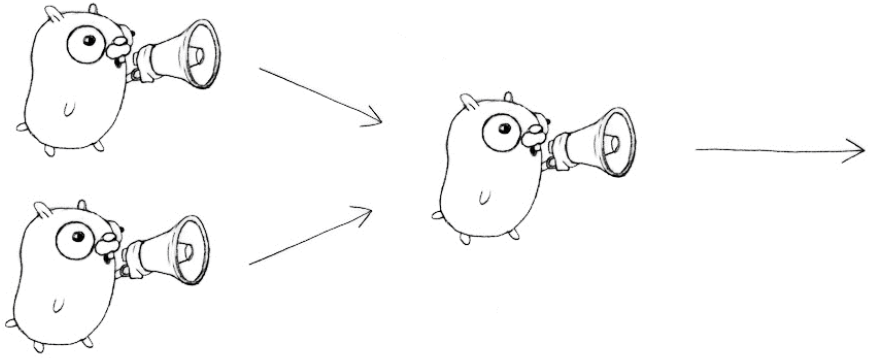
```go
add := func(
    done <-chan interface{},
    intStream <-chan int,
    additive int,
) <-chan int {
    addedStream := make(chan int)
    go func() {
        defer close(addedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case addedStream <- i+additive:
            }
        }
    }()
    return addedStream
}
```

```go
done := make(chan interface{})
defer close(done)
intStream := generator(done, 1, 2, 3, 4)
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
for v := range pipeline {
    fmt.Println(v)
}
```

Fan-out, fan-in pattern

- Problem: stages in a pipeline might be slower than the other and they might benefit from parallelism
  - Computationally intensive work
- Fan-out: start multiple goroutines to handle input from the pipeline
- Fan-in: combine multiple results into one channel





Fan-out

- Fan-out a stages of the processing if
  - It doesn’t rely on values that the stage had calculated before
  - It takes a long time to run
- No guarantee on the order concurrent copies run, nor in what order they return
  - A naive implementation of fan-out only works if the order in which results arrive is unimportant

```go
numFinders := runtime.NumCPU()
finders := make([]<-chan int, numFinders)
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}
```

Fan-out design

- Number of goroutines that are spined up matters
  - Use `runtime.NumCPU()` to find the number of OS threads that are used to run the goroutines
  - As a rule of thumb, fan-out `runtime.NumCPU()` goroutines, or profile your code to enhance the performance

Fan-in

- Involves *multiplexing* or joining together multiple streams of data into a single stream (merging)
  - Consumers read from the multiplexed channel
  - Spin up one goroutine for each incoming channel, and transfer the information from the multiple streams into the multiplexed stream

```go
fanIn := func(
    done <-chan interface{},
    channels ...<-chan interface{},
) <-chan interface{} {
    var wg sync.WaitGroup
    multiplexedStream := make(chan interface{})
    multiplex := func(c <-chan interface{}) {
        defer wg.Done()
        for i := range c {
            select {
            case <-done:
                return
            case multiplexedStream <- i:
            }
        }
    }
    // Select from all the channels
    wg.Add(len(channels))
    for _, c := range channels {
        go multiplex(c)
    }
    // Wait for all the reads to complete
    go func() {
        wg.Wait()
        close(multiplexedStream)
    }()
    return multiplexedStream
}
```

Fan-out, fan-in example

- Fan out to find prime numbers
- Fan in to print the results

*A bit too many goroutines for such a simple problem

```go
done := make(chan interface{})
defer close(done)
start := time.Now()
rand := func() interface{} { return rand.Intn(50000000) }
randIntStream := toInt(done, repeatFn(done, rand))
numFinders := runtime.NumCPU()
fmt.Printf("Spinning up %d prime finders.\n", numFinders)
finders := make([]<-chan interface{}, numFinders)
fmt.Println("Primes:")
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}
for prime := range take(done, fanIn(done, finders...), 10) {
    fmt.Printf("\t%d\n", prime)
}
fmt.Printf("Search took: %v", time.Since(start))
```

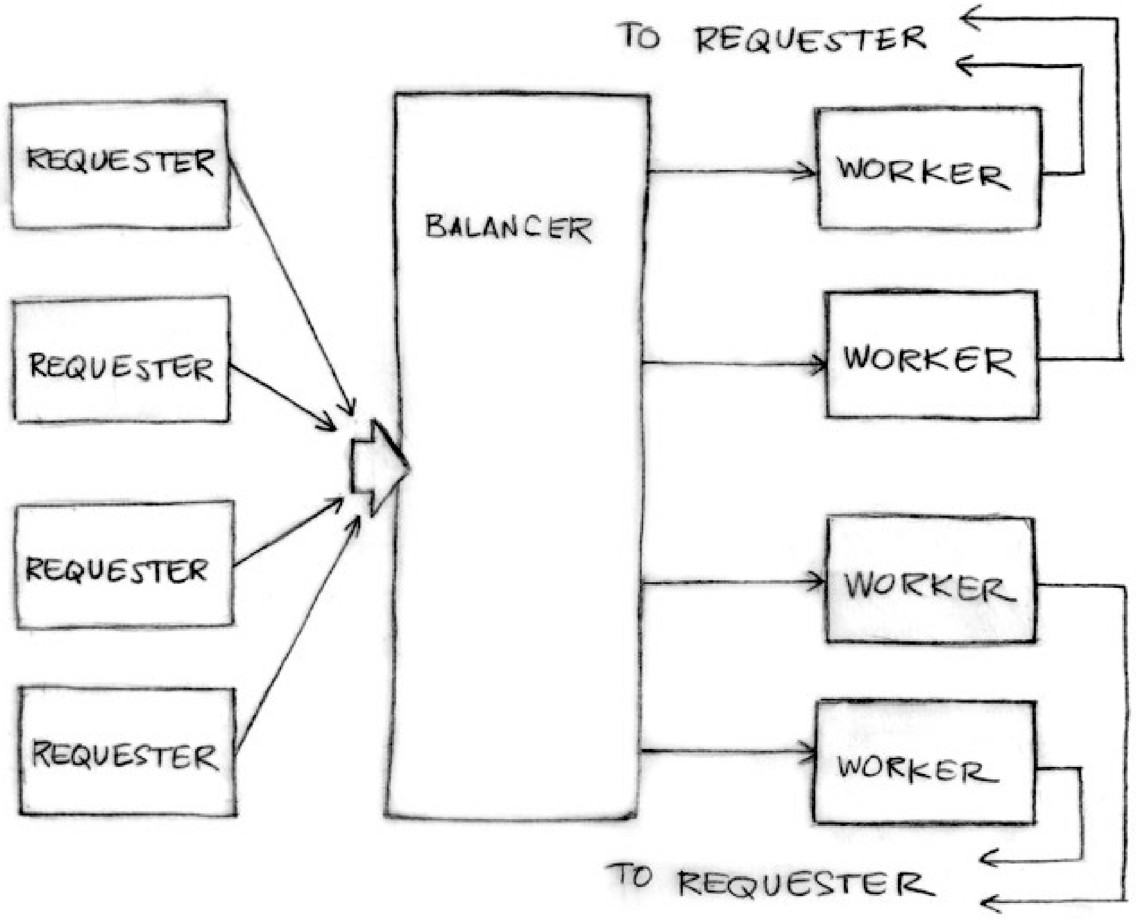
Outline

- Patterns
    - Confinement
    - For-select loop
    - Preventing goroutine leaks
    - Error handling
    - Pipeline
    - Fan-out, fan-in
- A load balancing example

A realistic load balancer (1)

- The requesters send Requests to the balancer
  - return channel inside the request

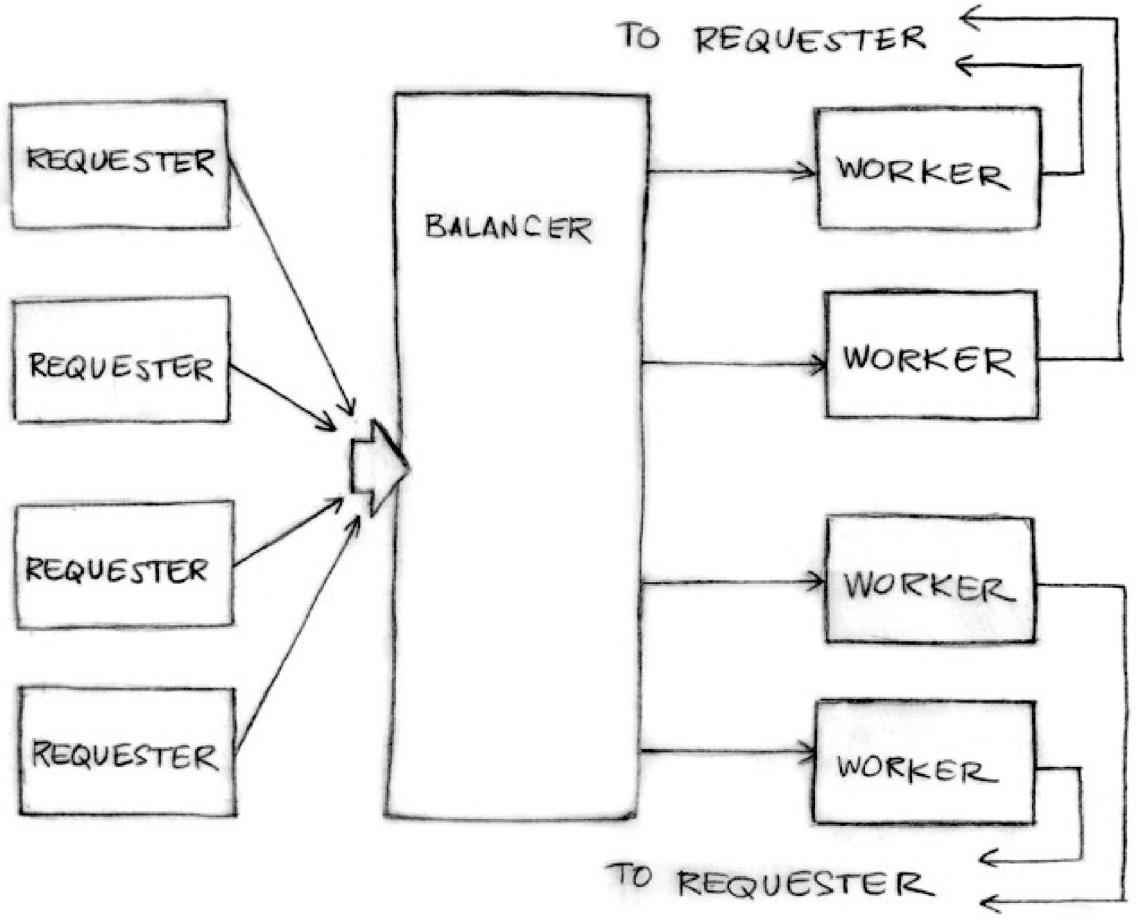
```
type Request struct {
    fn func() int  // The operation to perform.
    c  chan int    // The channel to return the result.
}
```



A realistic load balancer (2)

- The requester sends Requests to the balancer

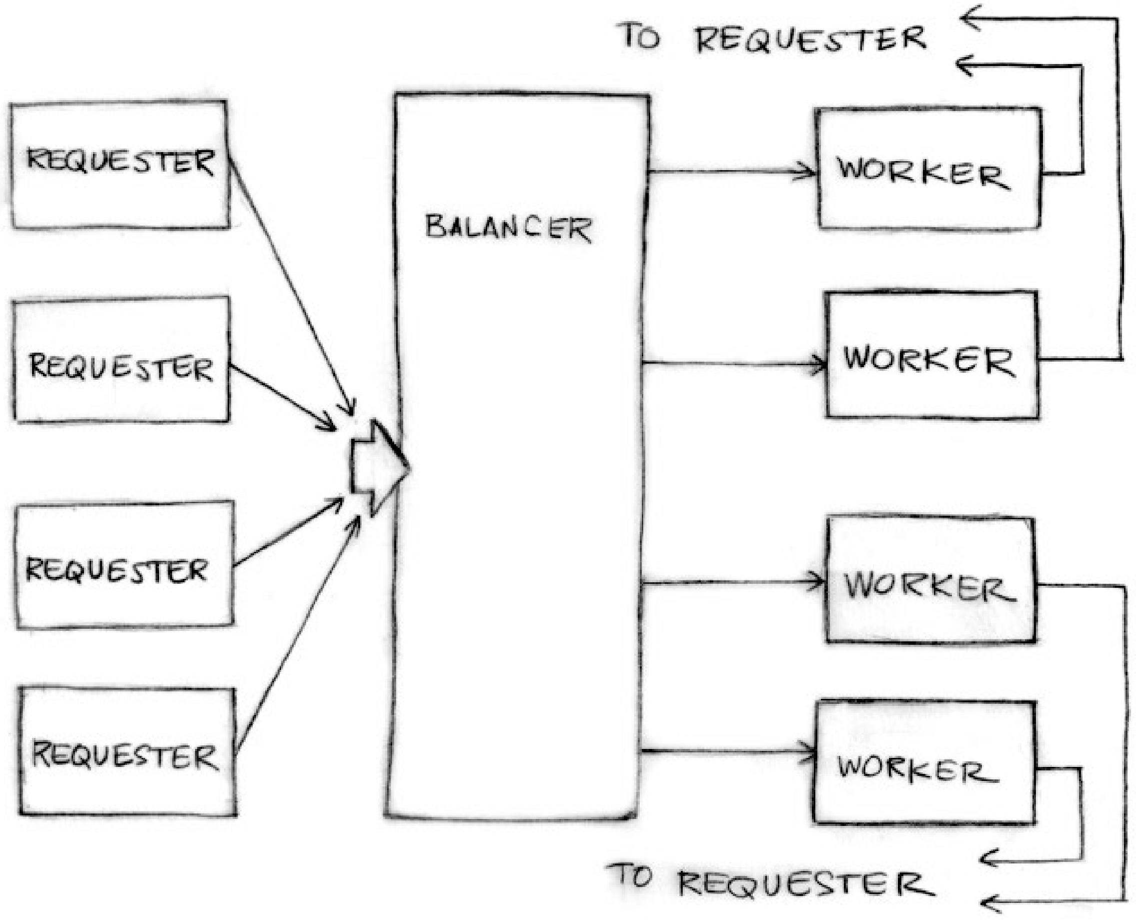
```
func requester(work chan<- Request) {
    c := make(chan int)
    for {
        // Kill some time (fake load).
        Sleep(rand.Int63n(nWorker * 2 * Second))
        work <- Request{workFn, c} // send request
        result := <-c              // wait for answer
        furtherProcess(result)
    }
}
```



A realistic load balancer (3)

- Worker definition
  - Channel of requests
  - Include load tracking data

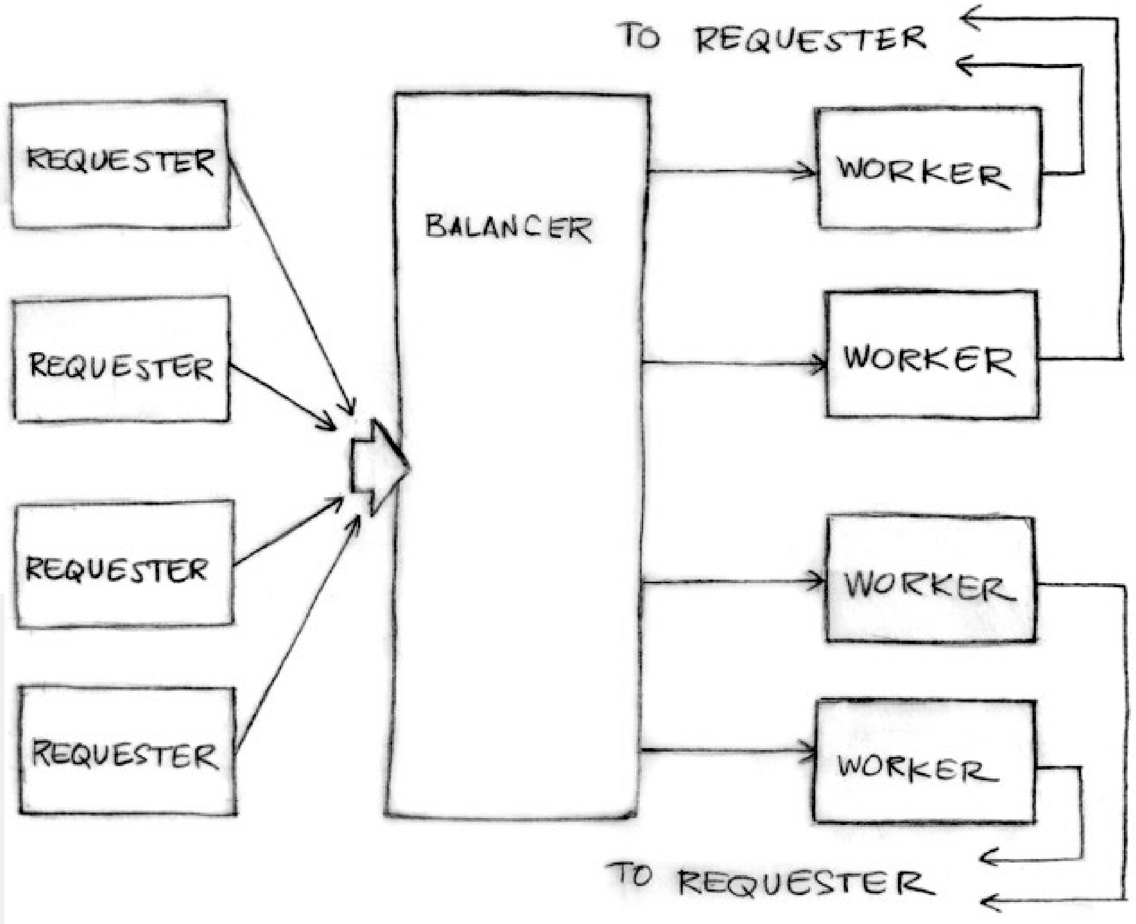
```
type Worker struct {
    requests chan Request // work to do (buffered channel)
    pending  int          // count of pending tasks
    index     int         // index in the heap
}
```



A realistic load balancer (4)

- Worker
  - The channel of requests (w.requests) delivers requests to each worker
  - The balancer tracks the number of pending requests as a measure of load
- Each response goes directly to its requester

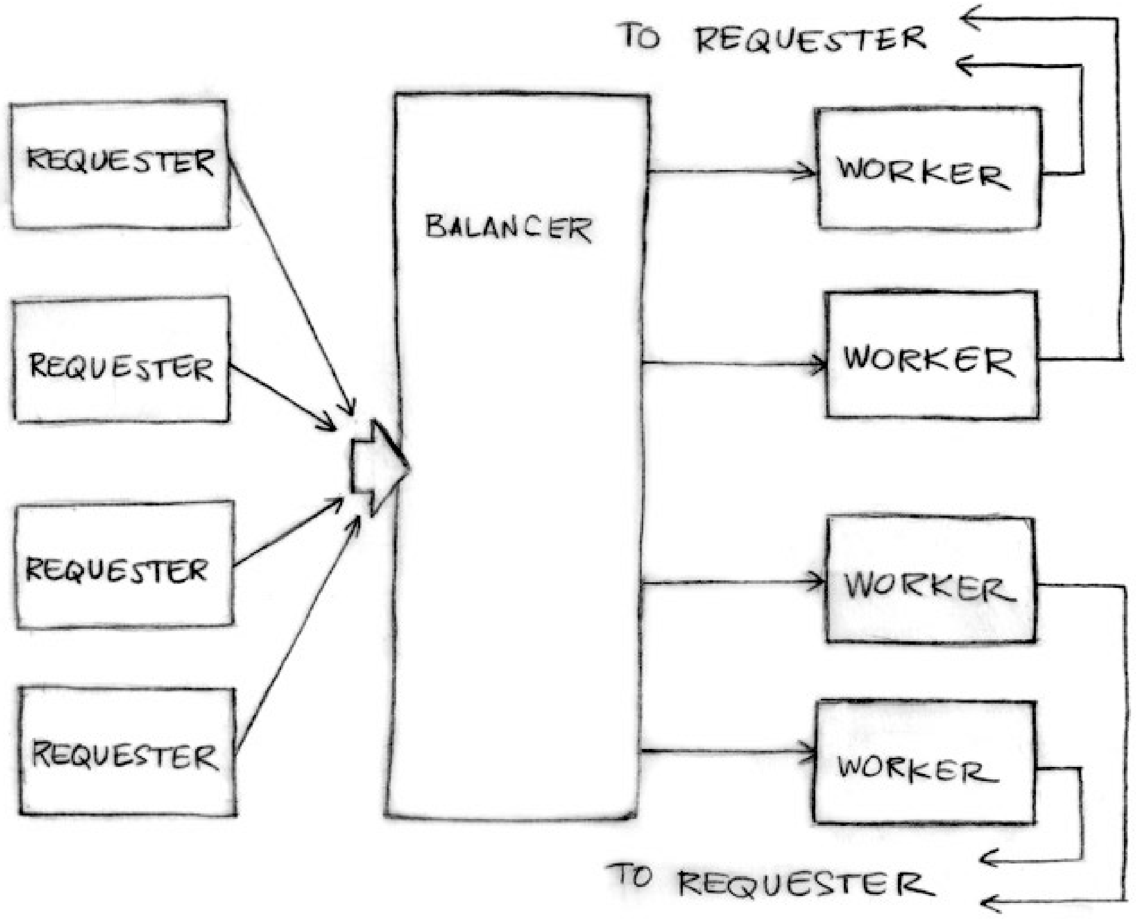
```
func (w *Worker) work(done chan *Worker) {
    for {
        req := <-w.requests // get Request from balancer
        req.c <- req.fn()   // call fn and send result
        done <- w           // we've finished this request
    }
}
```



A realistic load balancer (5)

- Balancer needs
  - A pool of workers
  - A single channel to which requesters can report task completion

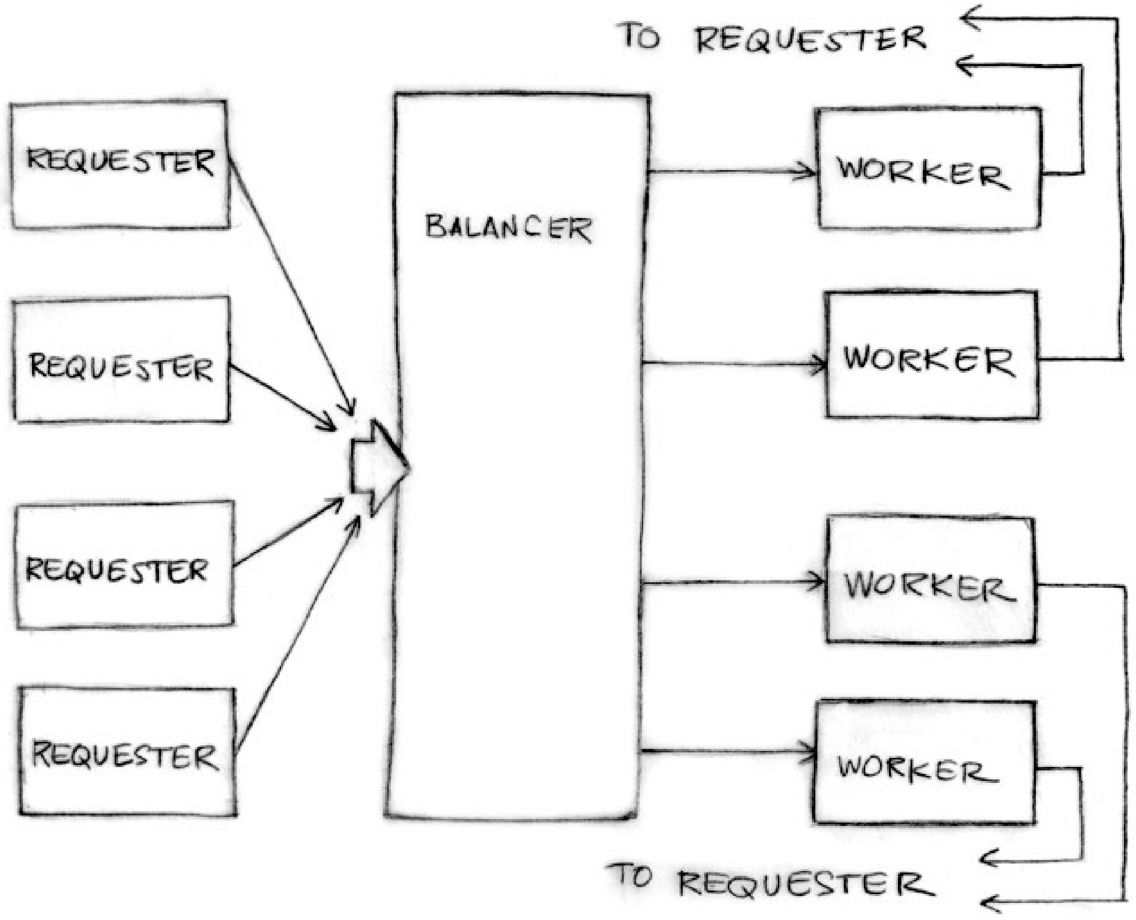
```
type Pool []*Worker

type Balancer struct {
    pool Pool
    done chan *Worker
}
```



A realistic load balancer (6)

- Balancer
  - Dispatches
  - Completes

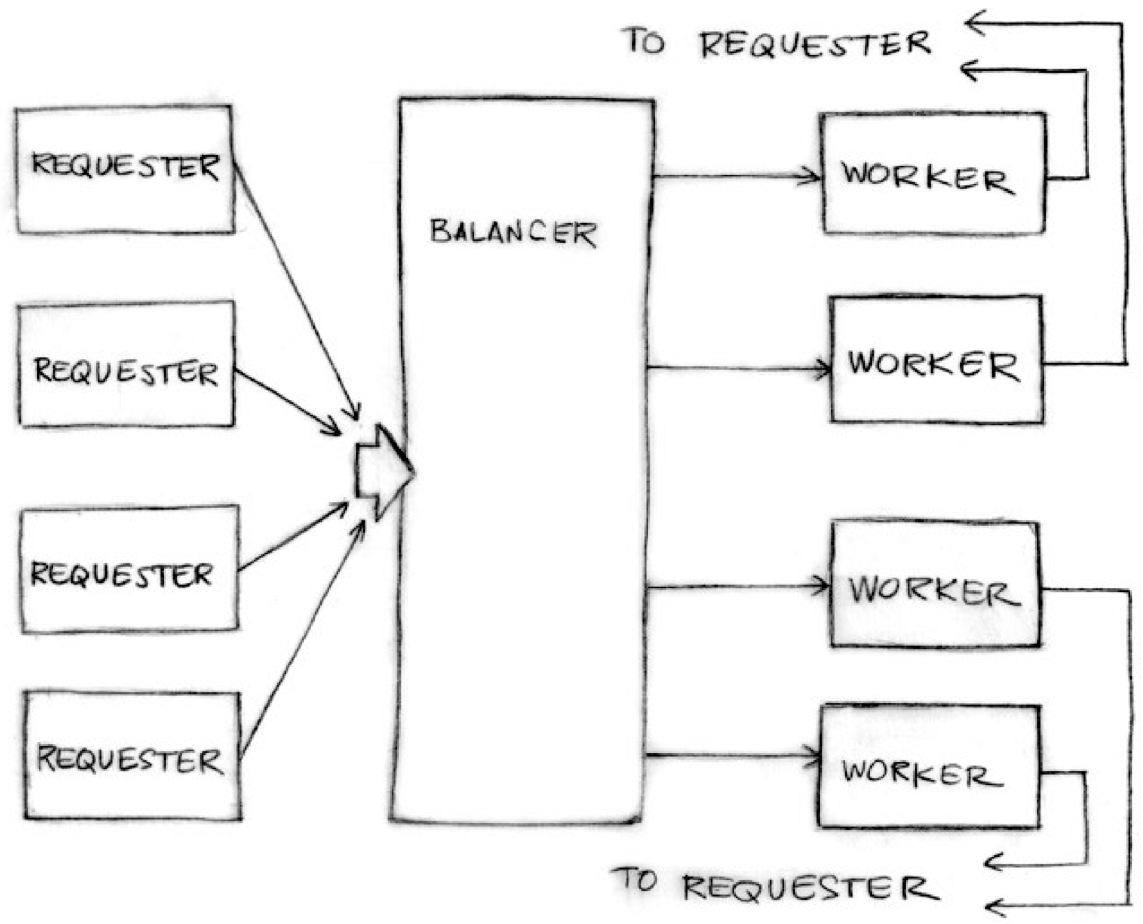
```
func (b *Balancer) balance(work chan Request) {
    for {
        select {
        case req := <-work: // received a Request...
            b.dispatch(req) // ...so send it to a Worker
        case w := <-b.done: // a worker has finished ...
            b.completed(w)  // ...so update its info
        }
    }
}
```



A realistic load balancer (7)

- A heap of channels – the Pool
  - An implementation of the Heap interface
  - Balance by making the Pool a heap tracked by load

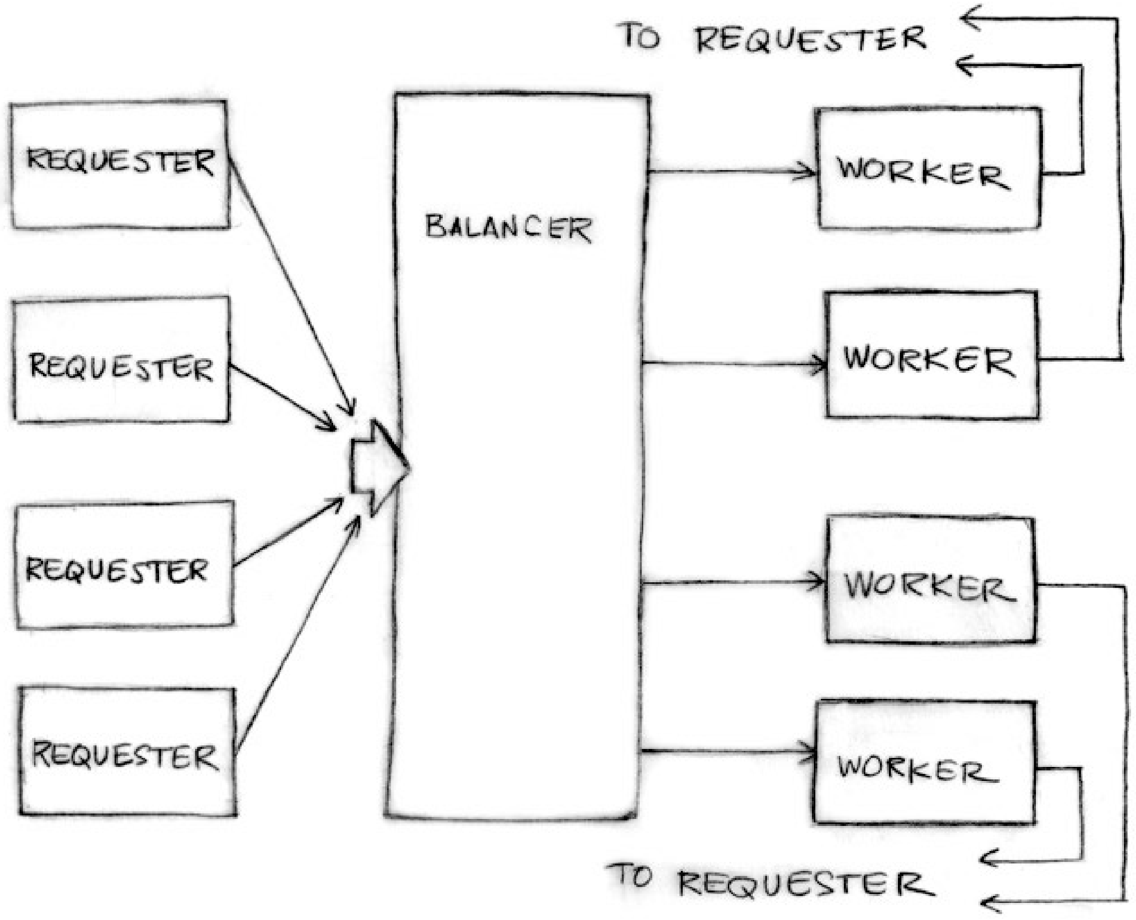
```
func (p Pool) Less(i, j int) bool {
    return p[i].pending < p[j].pending
}
```



A realistic load balancer (8)

- Dispatch

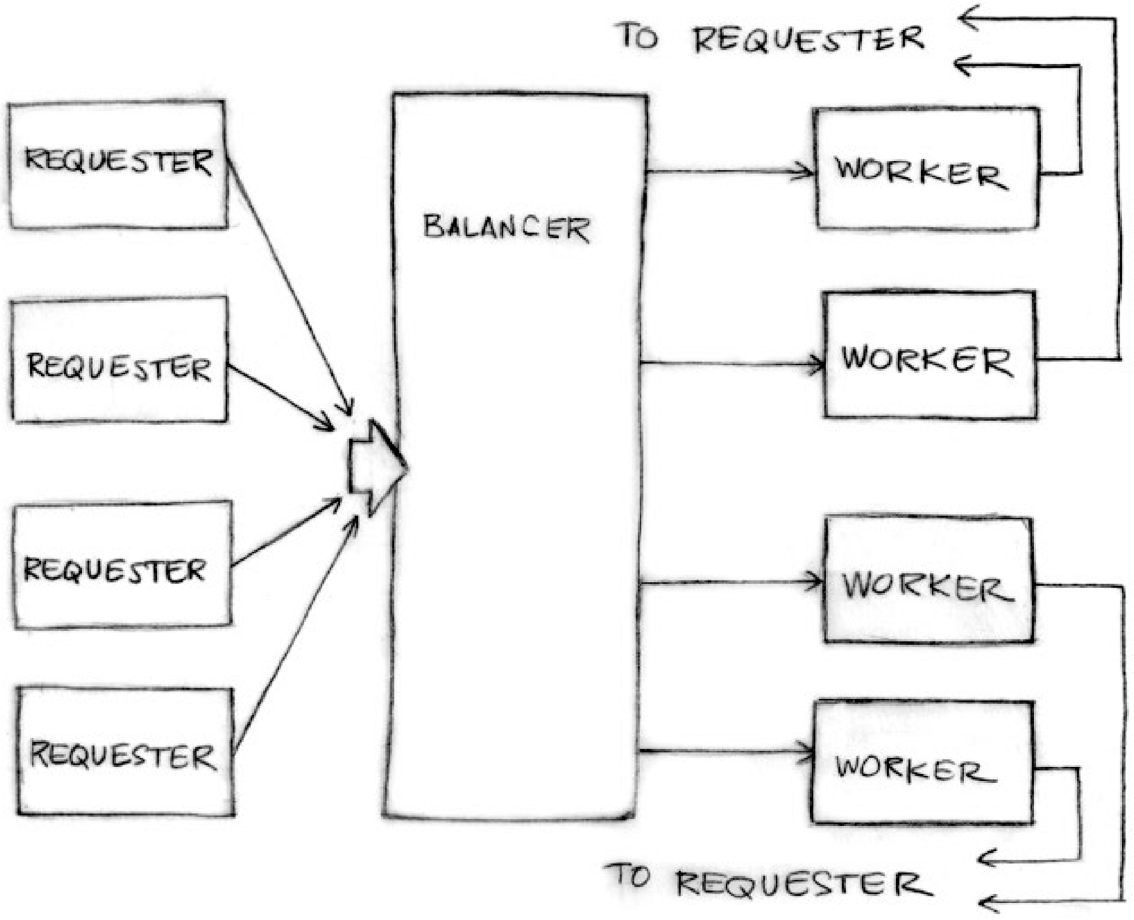
```
// Send Request to worker
func (b *Balancer) dispatch(req Request) {
    // Grab the least loaded worker...
    w := heap.Pop(&b.pool).(*Worker)
    // ...send it the task.
    w.requests <- req
    // One more in its work queue.
    w.pending++
    // Put it into its place on the heap.
    heap.Push(&b.pool, w)
}
```



A realistic load balancer (9)

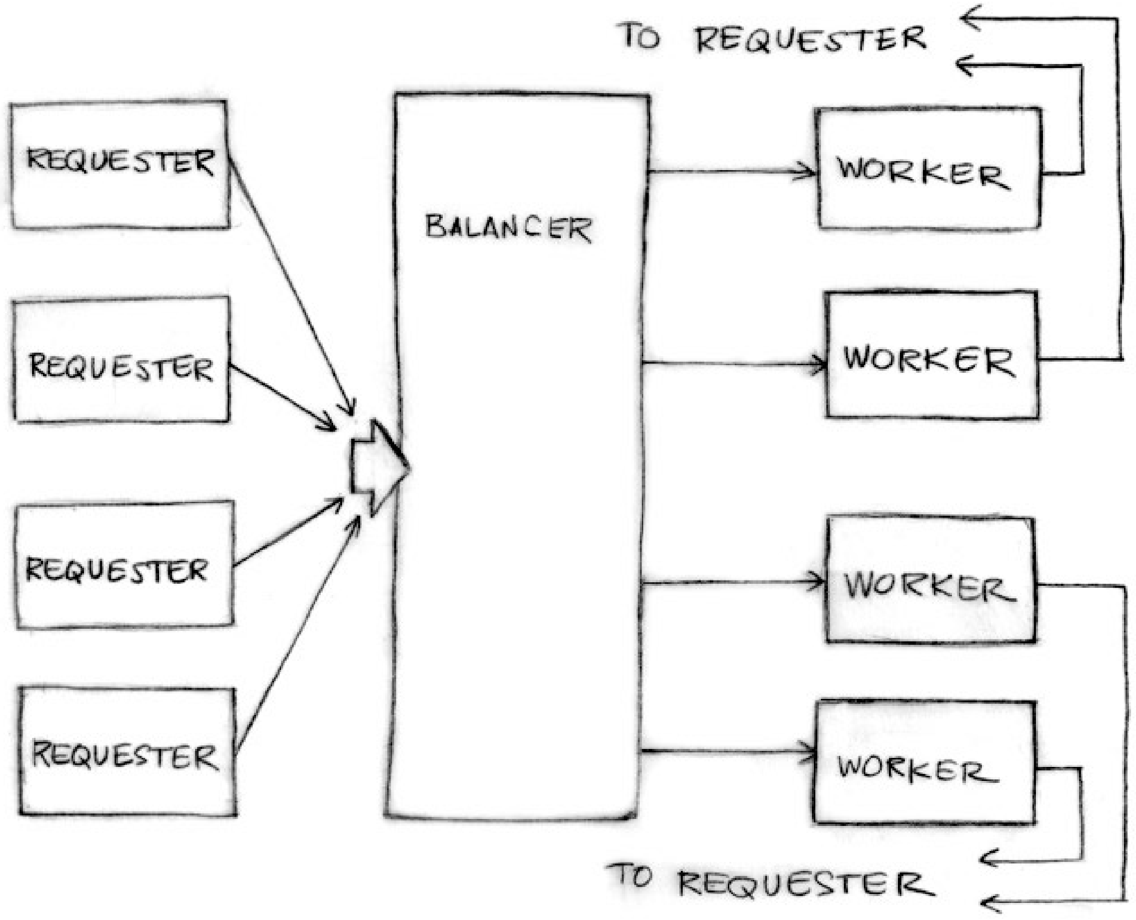
- Completed

```
// Job is complete; update heap
func (b *Balancer) completed(w *Worker) {
    // One fewer in the queue.
    w.pending--
    // Remove it from heap.
    heap.Remove(&b.pool, w.index)
    // Put it into its place on the heap.
    heap.Push(&b.pool, w)
}
```



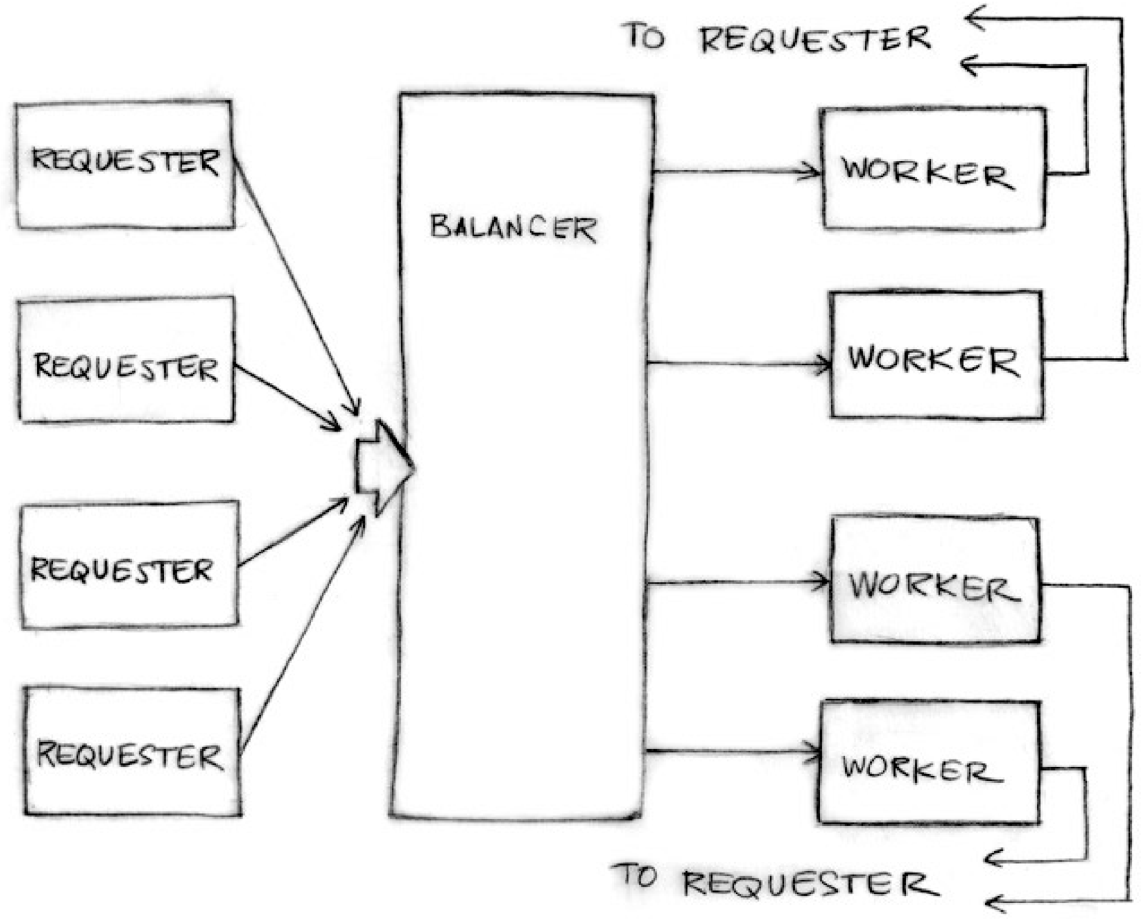
A realistic load balancer (10)

- A complex problem can be broken down into easy-to-understand components
- The pieces can be composed concurrently
- The result is easy to understand, efficient, scalable, and correct
- Decomposition allows for parallelism



Discussion

- How to enhance the load balancer?
- Will more concurrency translate into more performance?
- What patterns can we use in this problem?



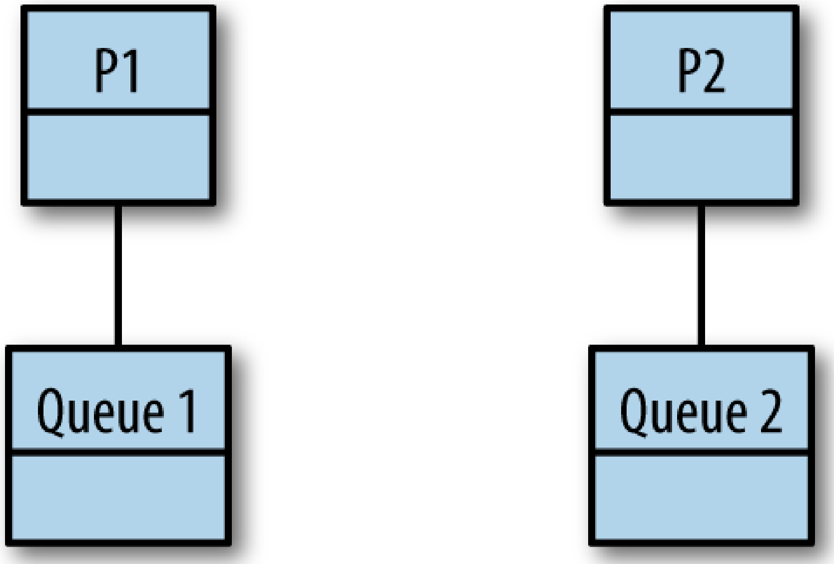
Summary

- We discussed patterns specific to Go
  - Enabled by channels and lightweight goroutines
  - Tradeoffs of using many goroutines/channels
  - Splitting the work helps us achieve more parallelism, but can be slower at times

- References
  - Concurrency in Go by Katherine Cox-Buday – chapter 4
  - Concurrency is not Parallelism by Rob Pike - https://talks.golang.org/2012/waza.slide

Bonus – Go Runtime

- Multiplexing goroutines onto OS threads - work stealing strategy
- Naïve strategies for sharing work
  - Fair scheduling - equally divide the tasks to the number of processors
  - Centralized queue with tasks
    - Locality
    - Imbalances
  - Decentralized work queues
    - Double ended queue: Deque



Work stealing

1. At a fork point, add tasks to the tail of the deque associated with the thread.

2. If the thread is idle, steal work from the head of deque associated with some other random thread.

3. At a join point that cannot be realized yet (i.e., the goroutine it is synchronized with has not completed yet), pop work off the tail of the thread's own deque.

4. If the thread's deque is empty, either:

   a. Stall at a join.

   b. Steal work from the head of a random thread's associated deque.

```go
var fib func(n int) <-chan int
fib = func(n int) <-chan int {
    result := make(chan int)
    go func() {
        defer close(result)
        if n <= 2 {
            result <- 1
            return
        }
        result <- <-fib(n-1) + <-fib(n-2)
    }()
    return result
}

fmt.Printf("fib(4) = %d", <-fib(4))
```

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine)	fib(4)		

Work stealing in action

T1 call stack	T1 work deque	T2 call stack	T2 work deque
(main goroutine)	`fib(4)`		

```go
var fib func(n int) <-chan int
fib = func(n int) <-chan int {
    result := make(chan int)
    go func() {
        defer close(result)
        if n <= 2 {
            result <- 1
            return
        }
        result <- <-fib(n-1) + <-fib(n-2)
    }()
    return result
}

fmt.Printf("fib(4) = %d", <-fib(4))
```

Stealing tasks or continuations?

- Stealing tasks:
  - an unrealized join point: the thread must pause execution and go fishing for a task to steal
  - *stalling join*

- In Go, goroutines are tasks.
- Everything after a goroutine is called is the continuation

- Enqueue and steal continuations instead of tasks

```go
var fib func(n int) <-chan int
fib = func(n int) <-chan int {
    result := make(chan int)
    go func() {
        defer close(result)
        if n <= 2 {
            result <- 1
            return
        }
        result <- <-fib(n-1) + <-fib(n-2)
    }()
    return result
}

fmt.Printf("fib(4) = %d", <-fib(4))
```

Stealing continuations

T1 call stack	T1 work deque	T2 call stack	T2 work deque
main			

```go
var fib func(n int) <-chan int
fib = func(n int) <-chan int {
    result := make(chan int)
    go func() {
        defer close(result)
        if n <= 2 {
            result <- 1
            return
        }
        result <- <-fib(n-1) + <-fib(n-2)
    }()
    return result
}

fmt.Printf("fib(4) = %d", <-fib(4))
```

In Go

- OS threads: T1, T2 (in the work stealing algo)
- Contexts: Number of processors (GOMAXPROCS) → work deque
- There are at least enough OS threads available to handle hosting every context: OS threads >= contexts

- Reference: https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html