CSC 330 | Team 1

TEST 3

CONCURRENCY WITH GO

Introduction

The Go programming language supports a variety of concurrency models. Traditional concurrency using mutexes and condition variables are available, but Go's preferred model of concurrency is message-passing, achieved using Go's "channel" data type. Go channels are based on a mathematical model known as CSP.

This paper compares Go and Petri Nets, explaining how concepts such as AND-split, AND-join, OR-split and OR-join could be implemented through Go programming constructs, namely channels, goroutines and select statements.

Go was designed with the explicit purpose of being pragmatic for software engineering. Thus it features many additions its channel model such as selective guards for inputs and outputs. CSP's limitation of one-to-one channels is not enforced, instead allowing one-to-many, many-to-one and many-to-many channel paradigms.

Furthermore, since channels are first class citizens in Go, they can be passed to functions as values and sent through other channels. Examples of Go constructs that exceed those of CSP are detailed.

Go's channels are explored in detail as we push the limits of what one can do with channels. Examples are given of how channels are passed between functions and goroutines. We then discuss sending channels over other channels, and the potential applications of this practice.

Go's most notable tool in its concurrency arsenal, goroutines, are expounded. An overview of the mechanics behind scheduling, load balancing and implementing these lightweight processes is presented. Moreover, the difference between single-core, multi-core, and distributed versions of Go is clarified.

Table of Contents

Question 1

Introduction

When designing a workflow using Petri Nets, there are cases where tasks must branch off from one another or a decision to execute a specific task must be made. The former is known as Parallel-AND and the latter is Selective-OR. Parallel routing involves splitting up tasks so they can be executed concurrently using an AND-split. After all tasks are completed, they join at the AND-join which synchronizes execution back into a singular flow.

Alternatively, selective routing chooses one or multiple tasks to be executed based on some precondition or decision using an OR-split. When these tasks have completed, an OR-join is used to continue synchronous workflow. Unlike parallel routing, there are multiple models for selective routing; some allow only one task to be executed while others may allow several. The following section discusses the concepts of parallel and selective routing, and how they can be implemented in Go using channels, goroutines and select statements.

Parallel-AND in Golang

An AND-split can be implemented in Go by creating two channels and two goroutines. Each goroutine will send a value into the channel. Both goroutines will want to be either executed at the same time or in some arbitrary order. A select statement implements the AND-join; select is used across both channels, waiting for both values simultaneously. At the resource [1.1], AND-join synchronizes the two parallel goroutines, allowing the code to continue after the goroutines have completed.[1.3]

In the document “Modeling Workflow with Petri-Nets” [1.2] by Wil van der Aalst and Kees van Hee, a model for simulating Parallel-AND and Selective-OR workflows is discussed. Figure 1.1, excerpted from [1.2], details this simulation. This example models handling incoming complaints for a hypothetical business. Upon first receiving a complaint, the workflow uses parallel processing to contact the client and notify the department. After completing both tasks, the data is gathered and then a decision is made regarding whether to reimburse the client or simply send a letter.

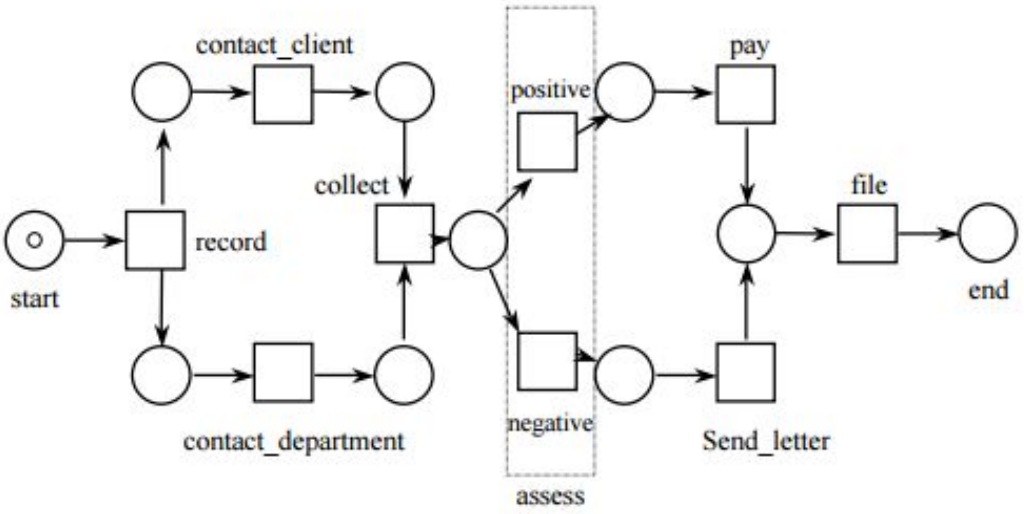

Figure 1.1: Example Model in Petri Nets using Parallel Routing and Selective Routing[1.2]

This model can be implemented in Go using a combination of goroutines and channels, as well as a select statement to ensure no further execution on the main thread is done until each goroutine has terminated. Below is an example implementation of the parallel routine for contacting the client and department.

```
package main

import "fmt"
import "time"

func main() {
    //Waiting for a complaint
    var complaint [2]string;
    complaint = receiveComplaint();

    ch1 := make(chan bool);
    ch2 := make(chan bool);

    //Here is the AND-split
    go contact_client(ch1, complaint);
    go contact_department(ch2, complaint);

    //This is where we AND-join
```

```
    for i := 0; i < 2; i++ {
            select {
            case msg1 := <-ch1:
             if (msg1) {
                 fmt.Println("Client contacted");
             } else {
                 fmt.Println("Failed to contact Client");
             }
            case msg2 := <-ch2:
             if (msg2) {
                 fmt.Println("Department contacted");
             } else {
                 fmt.Println("Failed to contact Department");
             }
         }
        }
}
//Pretend this function is waiting to receive an incoming complaint
func receiveComplaint() [2]string {
    return [2]string{"Client@domain.com", "ComplaintDescription"};
}
//Send response to the client
func contact_client(ch chan bool, complaint [2]string) {
    time.Sleep(time.Second * 2)
    ch <- sendEmail(complaint[0], "<html><b>We have received your
complaint blah blah ... </b></html>")
}
//Send complaint to department
func contact_department(ch chan bool, complaint [2]string) {
    time.Sleep(time.Second * 1)
    ch <- sendEmail(complaint[1], "<html><b>Complaint received: blah
blah ... </b></html>")
}
//Pretend this functions sends email to user
func sendEmail(email string, content string) bool {
    //Sends an email message and returns false if failed
    return true;
}
```

This code snippet uses two channels for each task that needs to be executed: one for each goroutine that will be created. In this example, the

goroutines will only write a boolean true or false into the channel; these values indicate whether the email was sent successfully or not. In the real life model, this channel would instead return a series of information that could later be used for the Selective-OR part of the workflow discussed in the next section.

Selective-OR in Golang

Selective routing has two main forms: explicit and implicit. Implicit routing is a fork in the workflow that executes the task that is most “eager” or is ready first (Figure 1.2). This means that each task executes and a decision as to whether the task will complete its execution is made at some point inside the routine. Explicit routing is much more deterministic: the decision to execute is made either at the beginning of the task or before creating it (Figures 1.3-1.4). In figure 1.5, each task’s execution is dependent on some precondition. Each task whose precondition is met will execute and then join as usual.

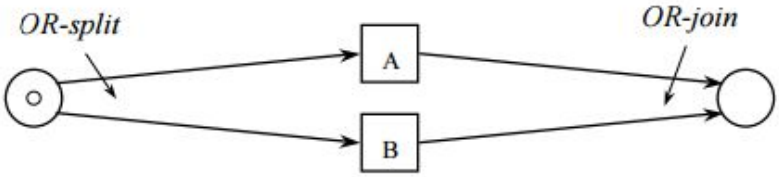

Figure 1.2: Implicit selective routing [1.2]

This form of selective routing can be implemented in Go through the use of goroutines and channels the same way that parallel routing was except with extra conditionals within the goroutine functions that cause termination if not satisfied. The main thread will ignore data in the channels for goroutines that terminate this way.

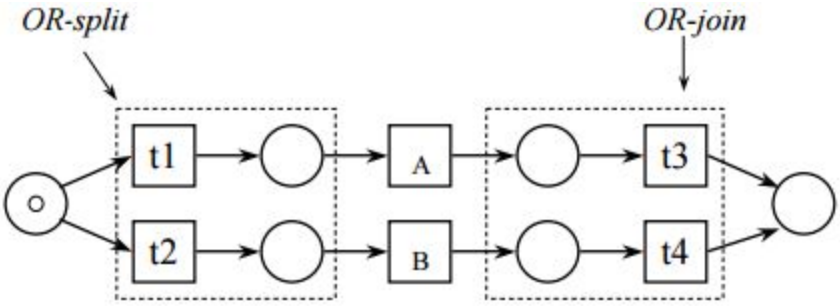

Figure 1.3: Explicit selective routing [1.2]

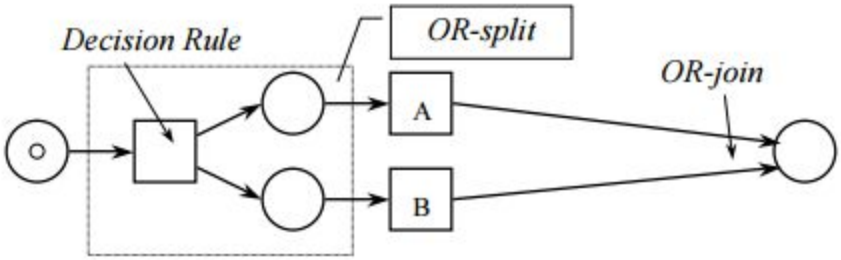

Figure 1.4: Explicit with case attributes: decision rule [1.2]

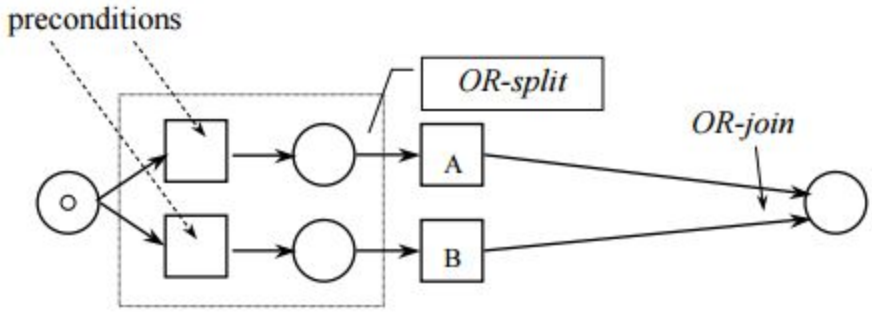

Figure 1.5: Explicit with case attributes: preconditions [1.2]

Each of these models of selective routing can be implemented in Go. For example, the use of selective routing found in figure 1.1 can be implemented using an OR-split where a condition is met using exclusive OR [1.3] (Figure. 1.4). A select statement may be used between two or more possibilities to implement the selective routing. There is a goroutine in each case of the select statements that implement the different tasks to be accomplished with only one task running based on the condition. Channels will be used to receive from the selected goroutine. This simulates the OR-join, completing the goroutine and allowing the code to continue.

```
    positive := make(chan bool);
    negative := make(chan bool);
    //Assume evaluate() is a function that decides on whether positive
or negative based on data from the first part. Say client_resp &
depart_resp is the data received from the first part of the model
    evaluate(client_resp, depart_resp, positive, negative);
    select {
    case msg := <-positive:
        if (msg) {
            go pay_client(positive);
```

```
        }
    case msg := <-negative:
        if (msg) {
            go send_letter(negative);
        }
    }
```

This select statement will either send a letter or pay the client based on a condition that is evaluated from contact client and contact department. Two separate goroutines are in their respective cases. Both have the possibility of being chosen, but only one goroutine will run.

Question 2

Introduction

Go has theoretical foundations based on the CSP model, so it follows naturally that Go would share CSP's channels. However, Go channels differ from the CSP model as CSP has only synchronous, unbuffered, and unidirectional communication. Additionally, Go further expands on many of the concepts introduced in CSP by incorporating more beneficial features such as the ability to use selective input and output guards and the ability to pass channels as values, which allows for multithreading, multiprocessing, and networking.

Selective Input and Output Guards in Go and CSP

In computer programming a guard is a boolean expression that, if true, allows a program to execute. This concept is especially useful in many languages, including Go, for implementing concepts like error handling. For example, in Go, a programmer may wish to create a function that finds the square root of all integers received from a channel. In this case, any negative integer is a valid output from the integer channel, but has no defined square root. A simple conditional if statement can be used as a guard to prevent the program from executing in this case.

CSP employs "guarded commands" [2.1] to implement this same idea. In CSP, a guarded command is expressed with the following syntax: G→S where G is a conditional statement and S is the statement that executes if and only if G is true. Using the example above, if a programmer wanted to find the square root of a value, they would first need to check if the value were non-negative as depicted with the following syntax: `x >= 0 → value := x`. This statement ensures x is nonnegative. If so, x is assigned to value, otherwise nothing happens.

While guarded commands are useful for creating complex programs, if a series of these commands are present in a command list and more than one are true, then only one command is arbitrarily chosen to execute while the others are discarded [2.1]. In Go, a select statement can be used to circumvent the randomness present in a CSP command list [2.2]. A select statement, as shown below [2.3], works by listing a set of cases in the form of conditional statements that are evaluated in the order in which they appear.

```
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
        }
    }
}
```

If one of the statements is true, that statement is executed and the others are discarded. Since the statements are evaluated in order, the ones with the highest priority are listed first.

One to Many, Many to One, and Many to Many Communication in Go

In Go, channels support not only one-to-one communication like CSP, but also one-to-many, many-to-one, and many-to-many communication [2.4]. In CSP, the only synchronization primitive is unidirectional sending or receiving on a channel. Furthermore, this communication is synchronous and unbuffered. Go's concurrency model no longer fits CSP if a mutex, semaphore, or condition variable is employed. Meanwhile, these variables are supported through Go's "share memory by communicating" model.

The primary difference of channels in Go and CSP is that in CSP, there can only be one sender or receiver on the channel's endpoints. In contrast, concurrent communication is modeled explicitly as channels in Go [2.5]. In other words, in CSP, "process A talks directly to process B," whereas in Go, goroutine A talks to channel C and goroutine B listens to channel C [2.6].

Moreover, CSP relies on explicit naming of source and destination processes. One-to-many, many-to-one, and many-to-many communication cannot be modelled in CSP because when a process issues a send() or receive() it must specify the name of the process to which the message is sent/received. It follows that when a process is required to receive messages from different senders, that process will wait for the first message arriving at that port [2.1].

The code below demonstrates how it is possible to have multiple writers sharing a channel with a single reader.

```
c := make(chan string)
```

```
for i := 1; i <= 5; i++ {
    go func(i int, co chan<- string) {
        for j := 1; j <= 5; j++ {
            co <- fmt.Sprintf("hi from %d.%d", i, j)
        }
    }(i, c)
}

for i := 1; i <= 25; i++ {
    fmt.Println(<-c)
}
```

Above, five goroutines write to a single channel five times and the main goroutine reads all of the twenty five messages [2.7].

Channels as Values

CSP ties communication to specific processes and sending messages directly to specific channels. Hoare, who originally presented the idea of CSP, hints at this possibility of multithreaded programming, but with the limitation that "each port is connected to exactly one other port in another process" [2.6]. This was a great influence to the Go programming language; however, Go can do more. It supports multithreading, multiprocessing, and networking.

Go models concurrent communication between actors as channels that pass signals and data. Channels communicate between goroutines and allow only one goroutine to have access to the data at a given time. One Go slogan says, “Do not communicate by sharing memory; instead, share memory by communicating”. [2.4] Go’s concurrency style is deterministic, and the passing message itself is the synchronizer, reducing the runtime and complexity.

Channels are defined differently in CSP and Go. In Go, channels are first-class citizens, which have the ability to communicate in any goroutine. CSP uses globally named channels that are not first class [2.8].

With Go it is possible to use a buffered channel completely within a single goroutine, which is not possible in CSP because it supports only one-to-one unbuffered channels. Below is an example of a buffered channel.

```
func main(){
      c := make(chan int, 1)
      c <-3
}
```

Question 3

Introduction

Channels are a means for goroutines to communicate with each other and synchronize execution. They are extremely valuable because they support the ability for programs to run multiple, smaller tasks at the same time. Using channels, goroutines are able to pass values to other goroutines and receive values from goroutines. Although channels are already useful, they can get far more advanced than that. Here we determine whether it's possible for channels to act as values (passed through functions as parameters or received from functions as return type), and if channels can be passed over other channels (resulting in new communication channels from existing ones).

Channels as Values

Passing channels as parameters in function calls is supported in Go and can be easily done using the following syntax.

```
func readFromChannel ( input <- chan type ) {

}
```

Similarly, it is possible for functions to have channels for return types. The following is an example of how this can be achieved [3.1].

```
// function returns a channel
func getChannel() chan bool {
     b := make(chan bool)
     return b
}
```

Although passing channels as parameters to functions and returning channels from functions may be interesting, it is very basic. A more important question is whether it is possible to pass channels over channels in Go.

Channels in Channels

You can indeed pass channels over channels in Go. Below is an example of a simple program which passes a channel of type "chan int" through another channel:

```
c := make(chan (chan int), 1)
a := make(chan int)
c<-a
print(<-c)
```

One reason for sending channels over channels may be if the sender needs to notify the receiver to perform some computation of the type before sending the results to another goroutine. The Go Blog discussed this in an article on pipelines in Go [3.2]. Though the article never passes a channel through a channel, it outlines how channels can be used to create multi-stage data pipelines.

For example, given a set of input channels and a set of "worker" goroutines, the work can be naturally distributed over the workers by setting up one "master" channel which serves input channels to workers. When a worker finishes the work supplied on the input channel, it gets another from the master channel. Below is a short illustrative program:

```
func doWork(v int) {
	fmt.Printf("%d", v)
}

func worker(master <-chan chan int) {
	c, master_ok := <-master
	for master_ok {
		value, ok := <-c
		for ok {
			doWork(value)
			value, ok = <-c
		}
		c, master_ok = <-master
	}
}

func main() {
	master := make(chan chan int)
	for i := 0; i < 10; i++ {
		go worker(master)
	}
	var wg sync.WaitGroup
	for i := 0; i < 100; i++ {
```

```
            producer := make(chan int)
            wg.Add(1)
            go func() {
                  for j := 0; j < 1000; j++ {
                        producer <- j
                  }
                  close(producer)
                  wg.Done()
            }()
            master <- producer
      }
      wg.Wait()
      close(master)
}
```

The above program creates 10 worker goroutines and 100 channels representing input from various sources. It then creates a new producer goroutine to send on each channel. When a producer is finished, it closes its channel to indicate that there will be no further input. The workers then compete for input channels by all receiving from the "master" channel.

Super-servers

Though the example above may seem contrived, it extends to the very real application of super-server daemons like `inetd`. `inetd` listens on a set of server ports for incoming traffic. When something comes in, it examines the port number and the type of traffic, then starts up another daemon or program (e.g. `httpd`, `ssh`) to handle it.

We can do the same in Go: imagine we seek to reimplement `inetd` (or something similar) in Go. Each port is wired to a channel in our Go program. The program waits for input from any of them using `select`. When it receives input, it creates a new goroutine of the appropriate type to handle the input, then passes that goroutine the entire active channel.

Note that the new goroutine will also need to notify the main goroutine when it is done. For this, the new goroutine should actually be passed another channel, one conventionally named "done". Here is a snippet of code for a hypothetical worker in our implementation:

```
      func httpWorker(c <-chan HttpRequest, done chan<- interface{}) {
      // Do important HTTP-related work. Imagine "doStuff"
```

```
        // returns false when the last packet is seen.
        for doStuff(<-c) { }
        close(done)
}
```

Closing a "done" channel to signal completion is a common pattern in Go, and is one of the primary reasons for passing channels between goroutines. With this pattern in place, goroutines can be safely trusted with a channel, then notify the main goroutine when they are finished with it.

Question 4

Introduction

Goroutines are lightweight threads that are managed by the Go runtime. These functions are unique in that they can run concurrently with many, possibly thousands, of other goroutines in the same program [4.1]. These are executed by using the *go* keyword as follows:

```
go functionName(parameters)
```

To manage all of the goroutines, the Go scheduler attempts to efficiently schedule goroutines to available system resources [4.2]. By assigning contexts to each system thread and allocating goroutines to each context, the scheduler can manipulate which threads manage which goroutines, allowing for it all to continuously run concurrently.

Implementation and Benefits of Goroutines

In order to make goroutines faster than threads used in other languages such as Java, several design choices were made. Primarily, goroutines are created and destroyed entirely by Go's runtime. This makes the creation and destruction of goroutines extremely cheap, and much quicker when dealing with many of them. Furthermore, since goroutines are lightweight by definition, their impact on stack memory consumption is very low, at approximately 2KB of space when they are first created. As a goroutine grows, the stack space it occupies dynamically changes, allowing for stack and heap storage to be allocated or removed as required. Finally, goroutines handle thread blocking more efficiently than threads. Instead of having the scheduler save *all* of the registers that were allocated to it as threads require, goroutines only call for having the program counter, stack pointer, and DX (data register) saved, resulting in a much lower cost [4.3].

Implementation of the Go Scheduler

For managing all of the goroutines in a program, Go implements a unique scheduler. To put it simply, each system thread holds a context (processor) which will each run a goroutine. By default, the number of contexts attached to system threads is the same value as GOMAXPROCS, which is "the value that controls the

operating system threads allocated to goroutines" [4.4]. Each context can run one goroutine at a time, however to preserve concurrency, they also have a queue where other called goroutines are placed while they wait for execution. Figure 4.1 shows this system using shapes: squares (M) for system threads, squares (P) for contexts, and circles (G) for goroutines.

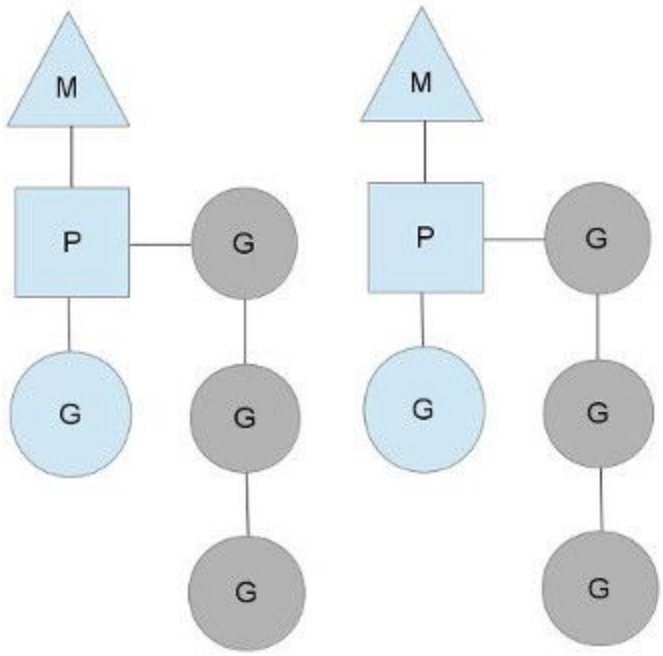

Figure 4.1 [4.7]

Each context will run their current goroutine until it stops; this can happen by I/O system operations, channel blocking, or a few other cases [4.5]. When this occurs, a new goroutine will be taken from the queue and executed. If a context runs out of goroutines on its queue, it will steal some from another context's queue and continue to run them concurrently. This is illustrated in Figure 4.2.

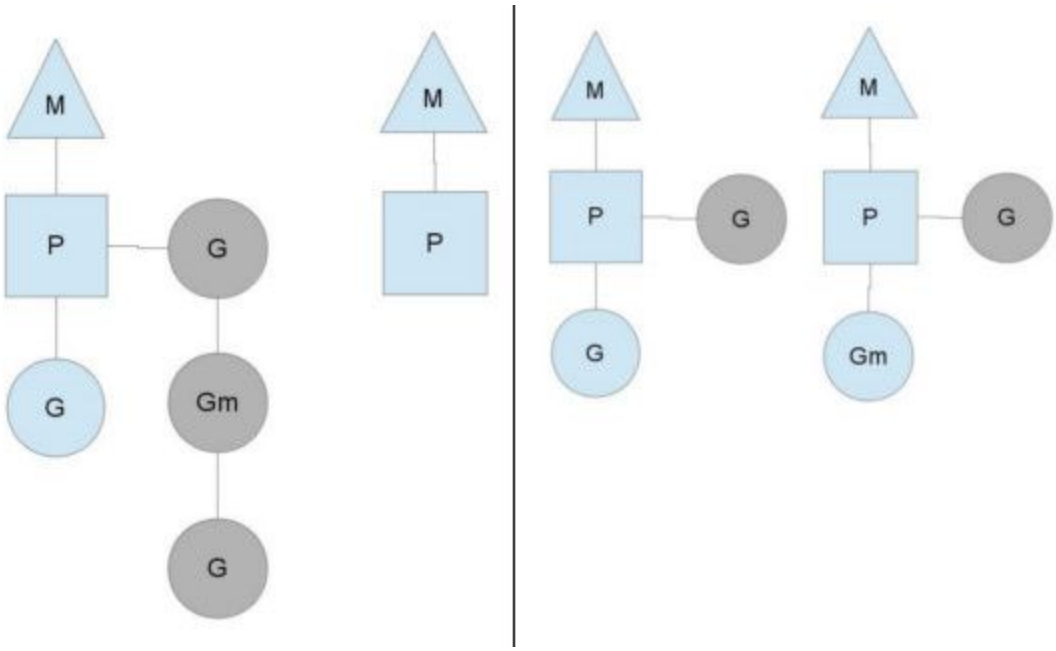

Figure 4.2 [4.7]

The left side of Figure 2 shows how one context is currently executing a goroutine and has three more queued up while another context is not executing anymore goroutines and has none queued. Instead of the thread releasing the context and going to sleep, it can take half of the goroutines from the others' queue and continue to execute them, speeding up the process considerably. In previous versions of Go, there was a global queue for goroutines as well and before stealing from goroutines in another context, the global queue would be checked for any goroutines that are executable [4.7].

Single-core vs. Multi-core vs. Distributed System Performance

Performance of Go programs varies depending on the size of the program, how much work it must do, and how much it relies on the use of channels between goroutines. In general, having a Go program execute on a single core will result in the faster runtime. This is primarily due to the fact that if a goroutine requires communicating with another via a channel and that goroutine is being held by another context on another system thread, it will have to switch contexts to communicate properly which comes at a significant cost in performance [4.8]. However, a tests have shown that if the workload becomes large enough, multiple cores can handle the processes more effectively than just a single core [4.9].

With regards to distributed systems, Go's performance is well known for being quite fast. This idea is reinforced by quote from a talk given by Rob Pike stating "the language was designed by and for people who write … large software systems". [4.10] While there are few resources detailing how well the language performs in these systems, it can be inferred from the languages core design concepts that it would perform well when scaled considerably.

Load Balancing in Go

Load balancing refers to efficient distribution of workload across computing resources, optimizing throughput, minimizing runtime, and to avoid overload. Below illustrates an example of how this might be implemented and executed in Go.

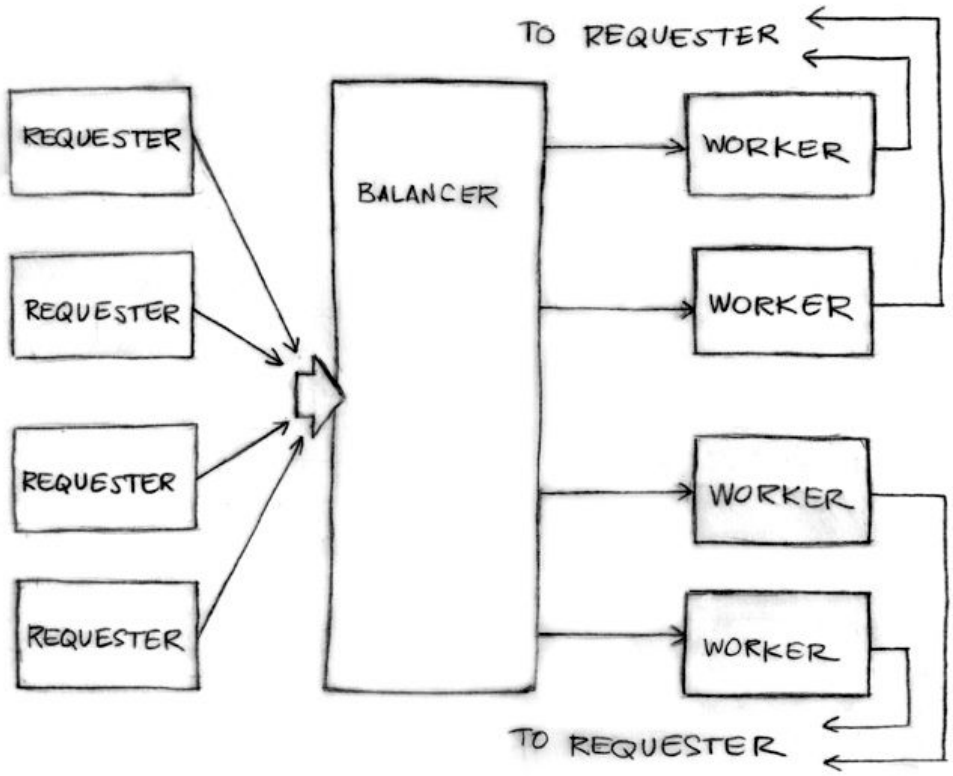

Figure 4.3: Visualization of a load balancer in Go [4.6]

Requesters will generate load and send requests to the load balancer:

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

Each worker has a channel of requests and keeps track of the number of tasks as well its location within a heap data structure that holds them [4.6].

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

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

The load balancer distributes the requests to the workers, and prioritizes the worker with the lightest load during execution. The requests channel sends requests to every worker while the balancer tracks the number of pending requests in order to evaluate the level of load [4.6].

```
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
    }
}
```

```
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)
}
```

The load balancer also requires a pool of workers and a channel from which requesters can determine when a request is completed. Once a task is completed, the heap is updated and the process repeats [4.6].

```
// 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)
}
```

Conclusion

Go is exceptionally good at its intended functions: concurrency and practical software development. It can support a form of AND-split, AND-join, OR-split and OR-joins similar to Petri Nets. It supports all the features of CSP and expands significantly on them, providing a flexible and powerful new form of communication among threads via message passing. Its channels are especially versatile, and can even be used to transmit other channels. Finally, Go scales exceedingly well, having been designed from the ground up for massive concurrency, whether on a highly multithreaded single machine or in a distributed system. Go's support for parallel processing is unrivalled, making it the obvious choice for implementing any system requiring a nontrivial amount of concurrency.

References

[1.1] M. McGranaghan. (2014, November 17). *Select*. Go by Example. Available: https://gobyexample.com/select

[1.2] G. Ferrari and W.M.P. van der Aalst. "Modeling Workflow with Petri-Nets", University of Pisa. Available: http://pages.di.unipi.it/ferrari/CORSI/SISD/Lezioni/WFModel.pdf

[1.3] W.M.P. van der Aalst. *The Application of Petri Nets to Workflow Management*. Journal of Circuits, Systems, and Computers, vol. 8, no. 1, pp. 21-66, 1998. Available: http://martinfowler.workflowpatterns.com/documentation/documents/vanderaalst98application.pdf

[2.1] C. Hoare, "Communicating Sequential Processes," The Queen's University, Belfast, 1978. Available: http://spinroot.com/courses/summer/Papers/hoare_1978.pdf

[2.2] Ø. Teig. (2012, 9 October). *Priority select in Go*. [Online]. Available: http://www.teigfam.net/oyvind/home/technology/047-priority-select-in-go/

[2.3] *Select*. A Tour of Go. [Online]. Available: https://tour.golang.org/concurrency/5

[2.4] T. Kappler. (2016, July). *Package csp*. GoDoc. [Online]. https://godoc.org/github.com/thomas11/csp

[2.5] (2015, 26 October). *Golang main difference from CSP-Language by Hoare*. Stack Overflow. [Online]. Available: http://stackoverflow.com/questions/32651557/golang-main-difference-from-csp-language-by-hoare

[2.6] A. Gerrand. (2012, July 13). *Share Memory by Communicating*. The Go Blog. [Online]. Available: https://blog.golang.org/share-memory-by-communicating

[2.7] (2015, July 14). *Multiple goroutines listening on one channel.* Stack Overflow. [Online]. Available: http://stackoverflow.com/questions/15715605/multiple-goroutines-listening-on-one-channel

[2.8] (2011, November). *Difference between CSP (channels) and coroutines (yield)*. Golang-nuts. [Online]. Available: https://groups.google.com/forum/#!topic/golang-nuts/Onswx7FpdxY

[3.1] A. Guz. (2013, December 6). *Golang Channels Tutorial*. [Online] Available: http://www.guzalexander.com/2013/12/06/golang-channels-tutorial.html

[3.2] S. Ajmani. (2014, March 13). *Go Concurrency Patterns: Pipelines and Cancellation*. The Go Blog. [Online]. Available: https://blog.golang.org/pipelines

[4.1] C. Doxsey. (2016). "Concurrency," in *An Introduction to Programming in Go*. [Online]. Available: https://www.golang-book.com/books/intro/10

[4.2] M. Dale. (2014). *Demystifying the Go Scheduler*. Absolute8511. [Online]. Available: http://www.slideshare.net/matthewrdale/demystifying-the-go-scheduler

[4.3] K. Sundarram. (2014, February 24). *How Goroutines work.* Krishna's blog. [Online]. Available: http://blog.nindalf.com/how-goroutines-work/

[4.4] *Package runtime*. The Go Programming Language. [Online]. Available: https://golang.org/pkg/runtime/

[4.5] I. Taylor. (2016, July 15). *How does the Golang scheduler work?* Quora. [Online]. Available: https://www.quora.com/How-does-the-golang-scheduler-work

[4.6] R. Pike. (2012, Jan 11). *Concurrency is not Parallelism*. [Online]. Available: https://talks.golang.org/2012/waza.slide#45

[4.7] D. Morsing. (2013, June 30). *The Go scheduler*. Morsing's Blog. [Online]. Available: https://morsmachine.dk/go-scheduler

[4.8] *Frequently Asked Questions (FAQ)*. The Go Programming Language. [Online]. Available: https://golang.org/doc/faq#Concurrency

[4.9] D. Muth. (2013, May 12). *Multi Core CPU Performance in Google Go*. Doug's Home On The Web. [Online]. Available: http://www.dmuth.org/node/1414/multi-core-cpu-performance-google-go

[4.10] R. Pike. (2012, Oct. 25). *Language Design in the Service of Software Engineering.* Go at Google. [Online]. Available: https://talks.golang.org/2012/splash.article