Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

— 1 —

Introduction

1.1 Remote Procedure Call (RPC)

This section will cover the concept of Remote Procedure Calls (RPCs) and how they are used in distributed systems and use the Go programming language to demonstrate such.

Establishing a Client-Server Connection

Definition 1.1: client-server model

The client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, **called servers**, and service requesters, **called clients**.

Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system.

Definition 1.2: Remote Procedure Call (RPC)

A Remote Procedure Call (RPC) is a protocol that allows a **client** computer request the execution of functions on a separate **server** computer.

RPC's abstract the network communication between the client and server enabling developers to write programs that may run on different machines, but appear to run locally.

Definition 1.3: RPC Call Stack

The RPC call stack facilitates communication between two systems via four layers:

1. **Application Layer:** The highest layer where the client application initiates a function call. On the server side, this layer corresponds to the service handling the request.
2. **Stub:** A client-side stub acts as a proxy for the remote function, **marshaling arguments** (converting them into a transmittable format) and forwarding them to the RPC library. On the server side, a corresponding stub, **the dispatcher**, receives the request, unmarshals the data, and passes it to the actual function.
3. **RPC Library:** The RPC runtime that manages communication between the client and server, ensuring request formatting, serialization, and deserialization.
4. **OS & Networking Layer:** The lowest layer, responsible for transmitting RPC request and response messages over the network using underlying transport protocols.

The request message travels from the client's application layer down through the stack and across the network to the server. The server processes the request in reverse, executing the function and returning the result to the client.

To illustrate the RPC call stack, observe the following diagram:

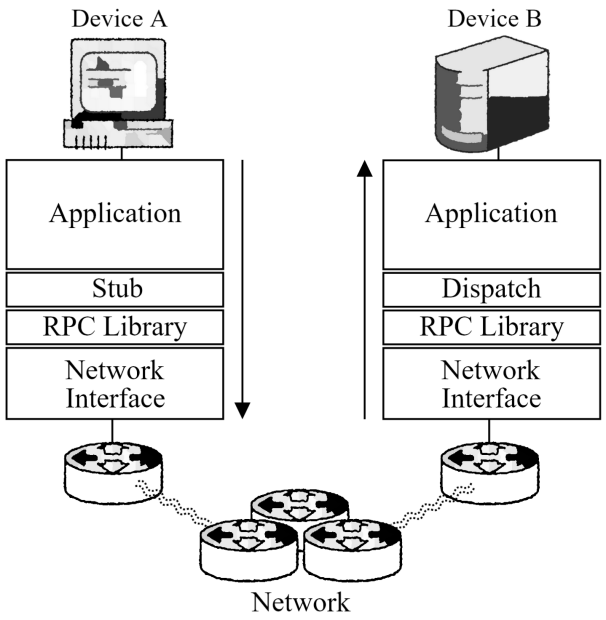

Figure 1.1: Client system A making a request to Server system B over RPC.

B runs through the stub and RPC library again to reply to A. In terms of time it might look like:

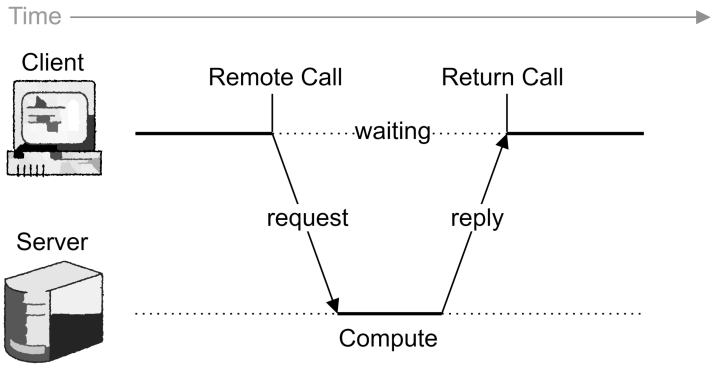

Figure 1.2: RPC call stack over time.

Once the client makes the call it waits for the server to process the request and return the result. The programmer need not worry beyond sending the request and receiving the response. The RPC deals with all the heavy work of facilitating the communication.

Now to discuss what marshaling and unmarshaling are:

Definition 1.4: Marshaling and Unmarshaling

Marshaling handles data format conversions, converting the object into a byte stream (binary data). This conversion is known as **serialization**. This is done as the network can only transmit bytes

Unmarshaling is the process of converting the byte stream into the original object called **deserialization**. This allows the server to process the request.

To illustrate serialization and deserialization, consider the following diagram:

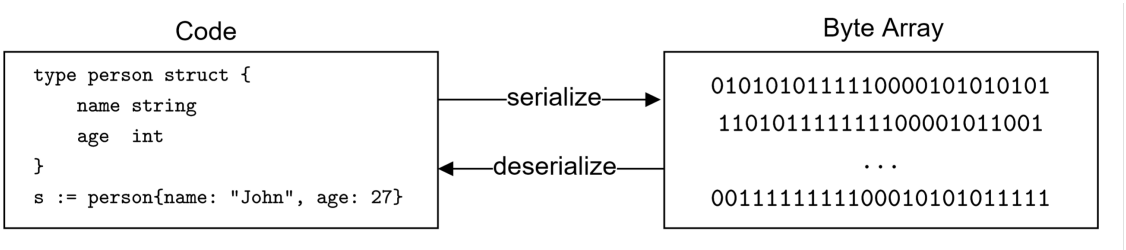

Figure 1.3: Serialization and Deserialization of data.

There is one cardinal rule to remember when dealing with RPCs:

Theorem 1.1: Network Reliability

The network is always unreliable.

That is to say, the network can drop packets, delay messages, or deliver them out of order. Anything that can go wrong will go wrong.

To handle network unreliability, we'll first consider two failure models:

Definition 1.5: At-least-once & At-most-once

- **At-least-once:** Regardless of failures, make the RPC call until the server responds. Works for read-only operations, otherwise, a strategy to handle duplicate requests is needed.
- **At-most-once:** Ensure the RPC call is made only once, even if the server fails to respond. This is done by having a unique identifier for each request. Each subsequent request tells the server which calls have already been processed.

For our communication to work *reliably* we need At-least-once and At-most-once with unlimited tries coupled by a fault-tolerant implementation. This brings us to the **GO RGC library**.

Definition 1.6: Go RPC Library

The Go RPC library provides a simple way to implement RPCs in the programming language Go. This gives us:

- At-most-once model with respect to a single client-server
- Built on top of single **TCP connection** (Transport Layer Protocol). This protocol ensures reliable communication between client and server.
- Returns error if reply is not received, e.g., connection broken (TCP timeout)

Now to discuss briefly how a basic TCP connection is made:

Definition 1.7: Establishing a TCP Connection (SYN ACK)

First a three-way handshake is a method used in a TCP/IP network to create a connection between a local host/client and server. It is a three-step method that requires both the client and server to exchange **SYN (synchronize)** and **ACK (acknowledgment)**.

1. The client sends a SYN packet to the server requesting to synchronize sequence numbers.
2. The server responds with a SYN-ACK packet, acknowledging the request and sending its own SYN request.
3. The client responds with an ACK packet, acknowledging the server's SYN request.

After the three-way handshake, the connection is established and the client and server can communicate exchanging SYN and ACK data-packets. To end the connection another three-way handshake takes placed, where instead of SYN, **FIN (finish)** is used.

Given this implementation, we approach somewhere in the realm of an **Exactly-Once model**:

Definition 1.8: Exactly-Once Model

The Exactly-Once model guarantees that a message is delivered exactly once to the recipient. Meaning, messages aren't duplicated, lost, or delivered out of order. However, In practice, data packets might do all of the above. Though with the right protocols in place, we can ensure order of logic is preserved.

Below we illustrate a simple TCP connection:

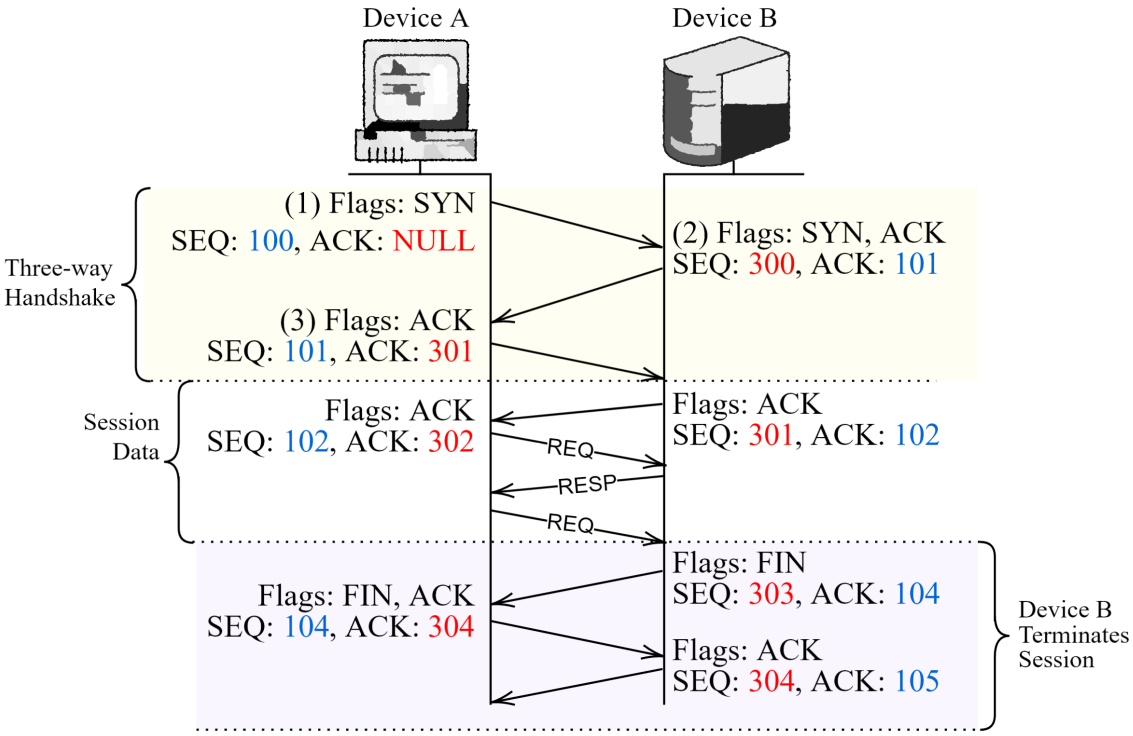

Figure 1.4: TCP Handshake, data transfer, and session termination.

Here the client (Device A) begins a three-way handshake with the server (Device B) to establish a connection. Both start with arbitrary sequence numbers for security purposes. With each packet received the two devices increment their sequence numbers accordingly.

Tip: If there still resides curiosity for the networking aspect of RPCs, consider reading our other notes: https://github.com/Concise-Works/Cyber-Security/blob/main/main.pdf

Asynchronous Function Calls

Let's begin to discuss how functions can run simultaneously using Go's **goroutines**:

Definition 1.9: Asynchronous Function Calls

An **asynchronous function call** is a function that executes independently of the main program flow, enabling tasks to run concurrently or in parallel.

Definition 1.10: Asynchronous Function Calls in Go: Goroutines

A **goroutine** is a **lightweight** (lower memory overhead and scheduling cost compared to traditional OS threads) concurrent execution thread in Go. Goroutines enable functions to run asynchronously. Unlike traditional operating system threads, goroutines are managed by Go's runtime.

A goroutine is created using the `go` keyword before a function call, signaling to the Go runtime to run the function asynchronously from the main program flow.

Definition 1.11: Go Runtime Scheduler

The Go runtime scheduler is responsible for managing goroutines via three conceptual entities:
The Go Scheduler: G, M, P

- **G (Goroutine):** A goroutine that holds the code to be executed.
- **M (Thread):** An OS thread that executes Go code via system calls or remains idle.
- **P (Processor):** Represents resources needed to execute code. The number of processors is determined by `GOMAXPROCS`.

If there are multiple goroutines, threads, and available processors, the scheduler matches them as follows:

- Many **Gs** (goroutines) are mapped to available **Ms** (OS threads), which execute them using **Ps** (processors) as execution resources.

Queues in the Scheduler

- **Global Run Queue (GRQ):** Holds all new goroutines that are yet to execute.
- **Local Run Queue (LRQ):** Holds goroutines that are assigned to a specific **P**.

For example, let the processors in the scheduler be defined as $P = \{P_1, P_2, \ldots, P_n\}$ where $n =$ `GOMAXPROCS`. Then the scheduler follows the following steps:

1. If P_1 has no more goroutines to execute, it follows these steps:
   a) Check **GRQ** for a **G** (goroutine) roughly *1/61th of the time*.
   b) If nothing is found, check **LRQ** again.
   c) If nothing is found, attempt to **steal** work from other **Ps**.
   d) If nothing is found, check **GRQ** one last time.
   e) Finally, **poll the network** (i.e., check for incoming network work).

To illustrate the Go runtime scheduler on a high-level, consider the following diagram in contrast to Figure **??**:

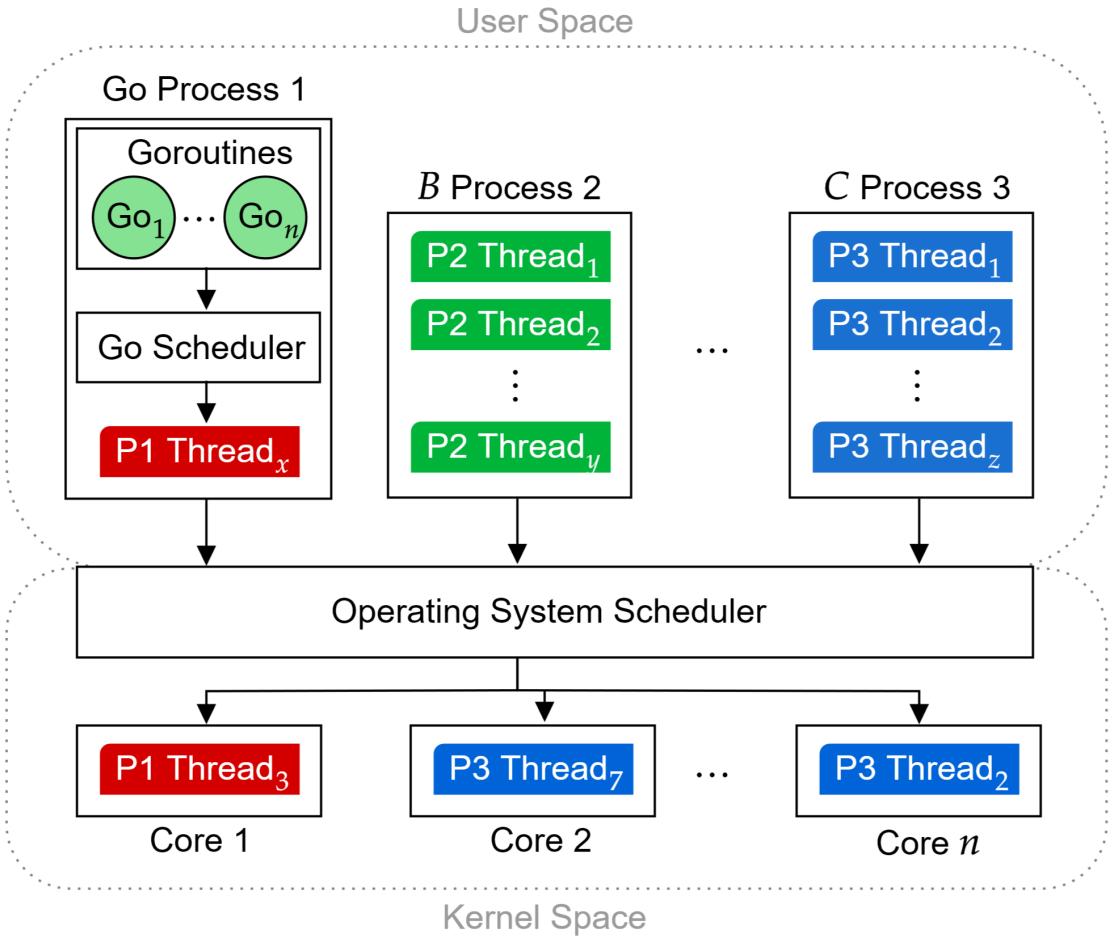

Figure 1.5: Go runtime scheduler with G, M, P entities.

Where the system has many different processes B, C and so on, we focus on the process that contains an instance of the Go runtime scheduler. Within this process each goroutine is scheduled by the Go runtime scheduler. The Go scheduler determines the number of threads needed and assigns goroutines to them. The process then presents these threads to the OS Scheduler which assigns them to the available cores.

In particular, **The OS has the final decision on which threads run on which cores.** The Go runtime scheduler only manages what threads to present. This is still helpful as the Go runtime can context switch the threads between goroutines before the handoff. Hence, the name **lightweight threads**, as context switching for the OS is expensive.

In particular we zoom in on the Go runtime scheduler of a single process:

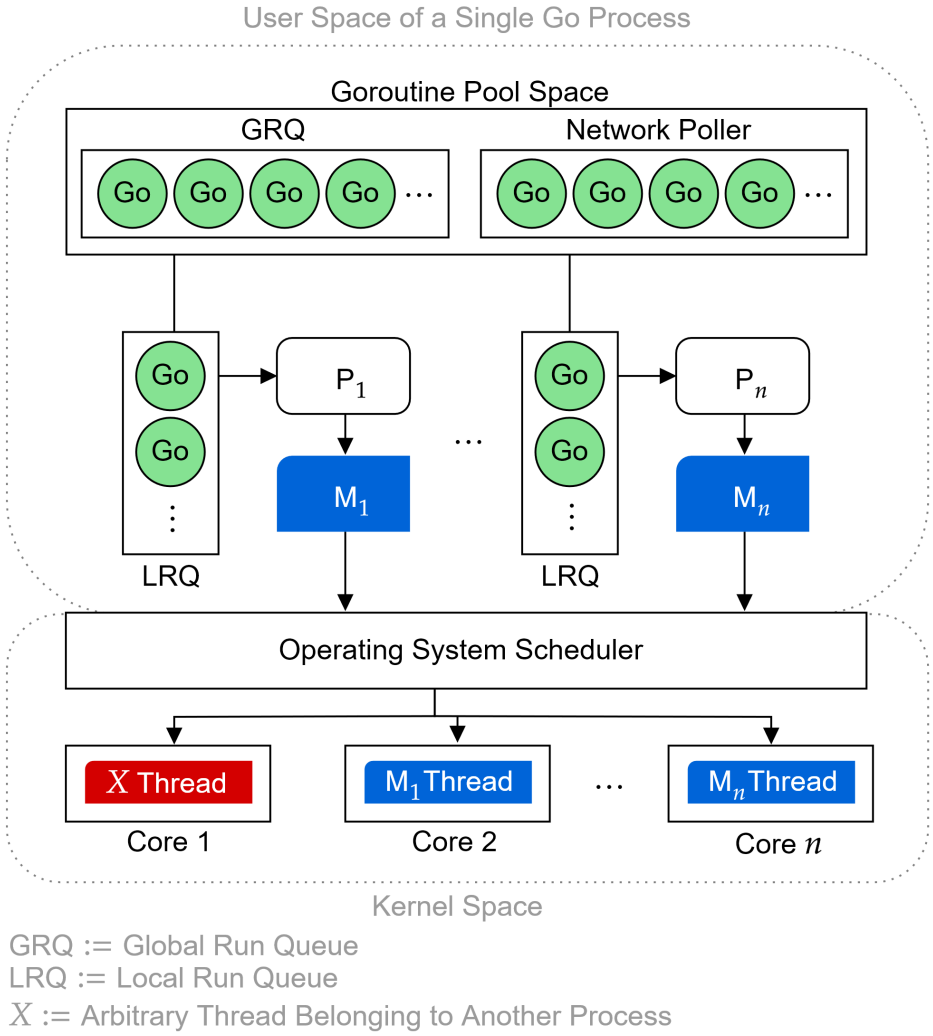

Figure 1.6: Go runtime scheduler within a single process.

Here our "Goroutine Pool Space" represents the latent goroutines waiting to be executed. We populate the LRQs of each P and their assigned M thread. The Ms are presented to the OS Scheduler. Moreover, the X is some arbitrary thread belonging to another process on the system. For emphasis

Theorem 1.2: Go Runtime Scheduler vs. OS Scheduler

The Go runtime only manages threads to present; The OS schedules threads to cores.

In all, the asynchronous nature of goroutines may create undefined behavior if not handled properly.

Example 1.1: Count to n using Goroutines

Consider the following Go program that counts to n using a goroutine:

Listing 1.1: Goroutine Example: Count to n

```go
package main // Required for Go programs to run as executables
import (
    "fmt"
    "time"
) // Import required packages : fmt for printing and time for sleep

// ''i:=1'' is short for ''var i int = 0''
func countUp(n int) {
    for i := 1; i <= n; i++ {
        // Anonymous function declared as a goroutine
        go func() {
            fmt.Println("Goroutine:", i)
        }()
    }
}
// Main entry point of the program
func main() {
    countUp(5)
}
```

However, this won't print anything as the main function exits before the independent goroutine can finish. A simple fix we'll do for now is put a sleep to wait for the goroutine to finish.

Listing 1.2: Adding a Sleep to Wait for Goroutine

```go
    func main() {
        countUp(5) // contains a goroutine
        time.Sleep(2 * time.Second) // Wait for goroutine to finish
        fmt.Println("Main function exits")
    }
```

Ensuring the main function waits for the goroutine to finish. ■

Theorem 1.3: Main Goroutine Thread

the main function of a goroutine is too a kind of goroutine. We may refer to it as the **main goroutine** or **main thread**. In particular, the main goroutine is the first to run and may finish before any other goroutine.

Though since calls happen independent of each other means they happen simultaneously.

Example 1.2: Count to n using Goroutines Corollary

Continuing off from the previous example (1.1), we'll get an output such as:

Listing 1.3: Output of Goroutine Example

```
...
func countUp(n int) {
    for i := 1; i <= n; i++ {
        go func() {
            fmt.Println("Goroutine:", i)
        }()
    }
}

func main() {
    countUp(5) // Runs countUp concurrently
    time.Sleep(2 * time.Second) // Wait for goroutine to finish
    fmt.Println("Main function exits")
}

/* Output:
Goroutine: 4
Goroutine: 3
Goroutine: 5
Goroutine: 2
Goroutine: 1
Main function exits
*/
```

The goroutine spawns multiple threads for each print of counter i. Therefore the order at which they execute is up to the Go runtime scheduler. ■

Theorem 1.4: Goroutines and Multithreading

Goroutines will attempt to run on multiple threads to achieve parallelism. However, if there isn't enough cores available, threads will run concurrently on the same core.

To declare how many cores can use, `runtime.GOMAXPROCS` from the `runtime` package can be used.

Listing 1.4: Setting the Number of Cores for Goroutines

```
import "runtime"
runtime.GOMAXPROCS(n) // n = number of cores to use
```

By default, Go will use the number of cores available on the machine.

Try these examples out in Go to get a feel for how goroutines work.

Definition 1.12: Installing and Running Go Programs

First, install Go from the official website: https://go.dev/doc/install. The Go file extension is `.go`:

- **To run a Go program:** Use the command `go run <filename>.go`.
- **To build a Go program:** Use the command `go build <filename>.go` to create an executable. Then run the program in a terminal via `./<filename>`.

Tip: This text will teach the necessary components as we go along. However, if one wishes to learn on their own a little first, consider the following resource: https://gobyexample.com/. Though this text does assume prior programming knowledge and should be follow-able without the resource.

Synchronization: Data Races & Deadlocks

Asynchronous functions introduces a problem: If two threads access the same memory location at the same time, we face corruption of data as they try to write over each other:

Definition 1.13: Data Race

A **data race** occurs when multiple threads or goroutines access the same memory location concurrently, and at least one of the accesses is a write operation, without proper synchronization. This leads to undefined behavior, including inconsistent data and unpredictable program execution.

To avoid data races we implement the following strategy:

Definition 1.14: Mutex (Mutual Exclusion)

A **mutex** (short for *mutual exclusion*) is a synchronization primitive that prevents multiple threads from simultaneously accessing shared resources. This allows a single thread to place a **lock** on the resource, ensuring exclusive access until the lock is released.

Go has their own mutex implementation:

Definition 1.15: Go Mutex

In Go, the `sync.Mutex` type provides a way to control access to shared data. A `Mutex` has two main methods:

- `Lock()`: declares that the current goroutine from which it resides has exclusive access to the resource.
- `Unlock()`: Releases the mutex, allowing other goroutines to access the resource.

Example 1.3: Increasing a Counter Variable with Goroutines

Consider the following example where a function `incCounter()` increments a shared counter variable:

Listing 1.5: Incrementing a Counter Variable

```
...
var counter int // declaring global counter variable

func incCounter() {
    counter = counter + 1
}

func main() {
    // forloop spawning an instance of incCounter() in a goroutine
    for i := 0; i < 1000; i++ {
        go func() {
            incCounter()
        }()
    }
    time.Sleep(5 * time.Second)
    fmt.Println("Counter:", counter)
}
/* Output: Counter: 982 */
```

By the end of the forloop, the counter will most often not be 1000. This is due to counter having a different state in each goroutine. To fix this, we'll use a mutex. So it is very possible that the first 2 goroutines look like this:

- Goroutine 1: `counter = 0 + 1`
- Goroutine 2: `counter = 0 + 1`

Where all three goroutines see the counter as 0, increment it all setting it to 1. ■

Now to fix the previous example (1.3) using a mutex:

Definition 1.16: Increasing a Counter Variable with a Mutex

To ensure a global variable counter is incremented correctly, we'll use a mutex:

Listing 1.6: Using a Mutex to Increment a Counter Variable

```
... // imported the "sync" package for the mutex
var counter int
var mu sync.Mutex // declaring a mutex

func incCounter() {
    mu.Lock() // Lock the mutex
    counter = counter + 1
    mu.Unlock() // Unlock the mutex
}

func main() {
    for i := 0; i < 1000; i++ {
        go func() {
            incCounter()
        }()
    }
    time.Sleep(5 * time.Second)
    fmt.Println("Counter:", counter)
}

/* Output:
Counter: 1000
*/
```

By using a mutex, we ensure that only one goroutine can access the shared counter variable at a time. **Important Note:** This does not ensure the order in which the goroutines run.

Though with mutexes may come another problem, what if a goroutine never releases the lock?

Definition 1.17: Deadlock

A **deadlock** occurs when two or more asynchronous processes are waiting for each other to release a resource, preventing all processes from progressing. This results in a program that hangs indefinitely.

In a large project a logical mistake in a sea of processes can lead to a deadlock.

Example 1.4: Deadlock Scenario

Say we have functions `task1()` and `task2()` that each require a mutex lock:

Listing 1.7: Deadlock Scenario

```
... // dots represent some passage of code

go func task1() {
    lockA.lock() ... lockB.lock()
    ...
    lockB.unlock() ... lockA.unlock()
}

go func task2() {
    lockB.lock() ... lockA.lock()
    ...
    lockA.unlock() ... lockB.unlock()
}

...
```

Depending on how the scheduler runs, these two tasks will lock each other out, halting the program indefinitely. ■

References & Pointers in Go

In Go, problems may arise from how Go deals with scoped variables:

Definition 1.18: Reference vs. Value Types

In Go, variables can be either **reference types** or **value types**:

- **Reference Types:** Point to a memory location where the actual data is stored. Changes to the reference type will affect all variables pointing to the same memory location.
- **Value Types:** Store the actual data in memory. Changes to a value type will not affect other variables.

Definition 1.19: Closures and Reference Types

In Go, if a variable isn't explicitly pass to a function, but is rather accessible from the function's scope, it is considered a **closure**. This closure is a reference to the variable, not the data itself.

Example 1.5: Closures and Goroutines

Let `data` be some channel and `do_something()` be some function that returns a value:

```
...
batch := 0
for i := 0; i < k; i++ {
    go func() {
        data <- do_something(batch)
    }()
}
batch++
...
```

Here, the `go func` closure will reference the `batch` variable, not the value. Hence the main program flow (the main thread) might increment `batch` before the goroutine runs, leading to undefined behavior. To fix this, we pass the variable as an argument to the goroutine:

```
...
batch := 0
for i := 0; i < k; i++ {
    go func(batch int) {
        data <- do_something(batch)
    }(batch)
}
batch++
...
```

Now the goroutine will receive the value of `batch` at the time of the loop iteration. ■

Many data-structures in Go pass by value. Pointers ensure we are updating the original object:

Definition 1.20: Passing Pointers in Go

Pointers in Go pass the memory address of a variable via the `&` operator. To access the value stored at the memory address, use the `*` operator. E.g.,

```
var x int = 5
var y *int = &x // y stores the memory address of x
fmt.Println(*y) // Prints the value stored at the memory address
```

Waiting for Goroutines to Finish

Previously we used `time.Sleep()` to wait for goroutines to finish; However, Go provides a solution to this problem:

Definition 1.21: Wait Groups

A **wait group** is a synchronization primitive in Go that allows the main program to wait for a collection of goroutines. A wait group is a counter spawned with `sync.WaitGroup` and has three main methods:

- `Add(n int)`: Increments the wait group counter by `n`.
- `Done()`: Decrements the wait group counter by 1.
- `Wait()`: Blocks the main program until the wait group counter reaches 0.

Example 1.6: Using Wait Groups

Let's consider the following example where we use a wait group to wait for goroutines to finish:

```
...
var wg sync.WaitGroup // declaring a wait group
var mu sync.Mutex
for i := 0; i < 1000; i++ {
    wg.Add(1) // Increment the wait group counter
    go func() {
        mu.Lock()
        incCounter()
        mu.Unlock()
        wg.Done() // Decrement the wait group counter
    }()
}
wg.Wait() // Wait for wg counter to reach 0
```

■

An aside on a handy feature of Go:

Definition 1.22: Deferred Function Calls

In Go, the `defer` keyword **defers a function call** to run at the end of the innermost scoped function. Deferred functions are often used to ensure cleanup tasks are executed, such as closing files or releasing resources.

Example 1.7: Deferred Function Calls

Consider the previous example (1.6) with a deferred function call of `wg.Done()`:

```
    ...
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done() // Deferred function call
            mu.Lock()
            incCounter()
            mu.Unlock()
        }()
    }...
```

The `wg.Done()` is deferred until the goroutine completes. ■

Sending Messages Between Goroutines

Now, say there are tasks A and B, for which B depends on the completion of A. Since A and B both run independently, we need a way for B to wait for a signal from A. This is where **channels** come in:

Definition 1.23: Channels

A **channel** is a typed conduit through which goroutines communicate. Channels allow goroutines to send and receive data. Channels are created using the `make()` function with the `chan` keyword.

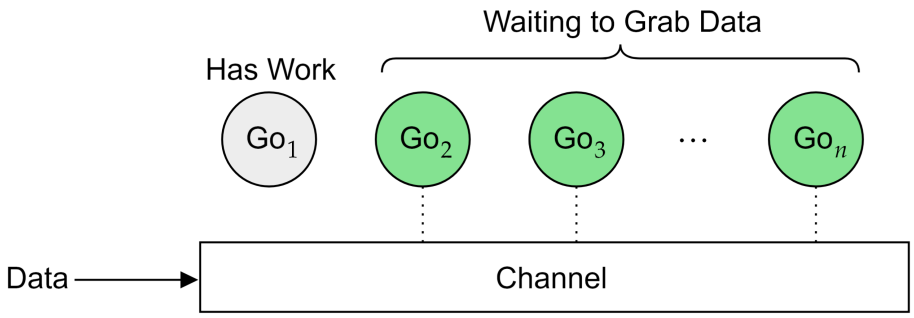

Figure 1.7: A collection of Goroutines competing for the next channel resource.

Note, despite the diagrams order of goroutines, the order in which they run is up to the Go runtime scheduler.

Example 1.8: Synchronizing Incoming Data Processing

Consider the following example where we use a channel to synchronize the processing of incoming data:

Listing 1.8: Using Channels to Synchronize Downloading and Processing

```
package main

import (
    "fmt"
    "sync"
    "time"
)

// Download function simulates downloading data
func download(wg *sync.WaitGroup, i int) {
    defer wg.Done()
    fmt.Printf("Downloading: Resource_%d...\n", i)
    time.Sleep(5 * time.Second) // Simulating download time
    fmt.Printf("Download complete: Resource_%d\n", i)
}

// Process function simulates processing data
func process(wg *sync.WaitGroup, i int) {
    defer wg.Done()
    fmt.Printf("Processing: Resource_%d...\n", i)
    time.Sleep(1 * time.Second) // Simulating processing time
    fmt.Printf("Processing complete: Resource_%d\n", i)
}

func main() {
    var wg sync.WaitGroup

    // Simulating downloading and processing 5 resources concurrently
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go download(&wg, i)
        wg.Add(1)
        go process(&wg, i)
    }

    wg.Wait()
    fmt.Println("Main function exits")
}
```

■

Theorem 1.5: Channel Types

In Go, channels can be either **unbuffered** or **buffered**:

- **Unbuffered Channels:** Require a sender and receiver to be ready to communicate. If the receiver is not ready, the sender will block until the receiver is ready.
- **Buffered Channels:** Allow a sender to send data to a channel without the receiver being ready. The channel will store the data until the receiver is ready.

Unbuffered channels undergo a **handshake** process:

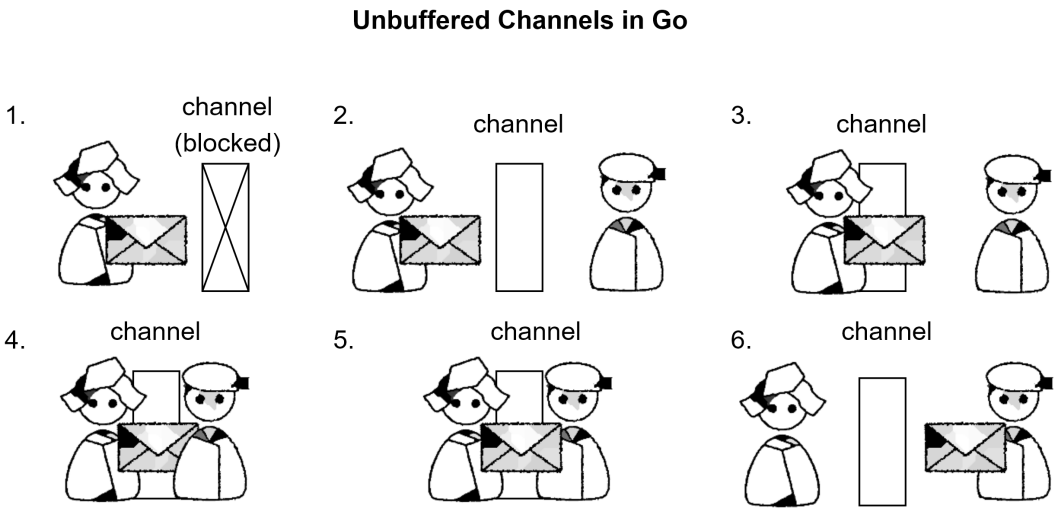

Figure 1.8: Handshake process of an unbuffered channel.

Say Alice (left) want to send a message to Bob (right) over a channel. (1) The channel is blocked until Bob is ready to receive. (2) The channel is no longer blocked read for the exchange. (3) Alice preforms a **send** and is locked into the operation until Bob **receives** the message. (4-5) Bob enters the channel and receives the message; Both Alice and Bob are locked into the operation until the message exchange is complete. (6) they both are free to continue their operations.

In contrast buffered channels allow for a more asynchronous approach:

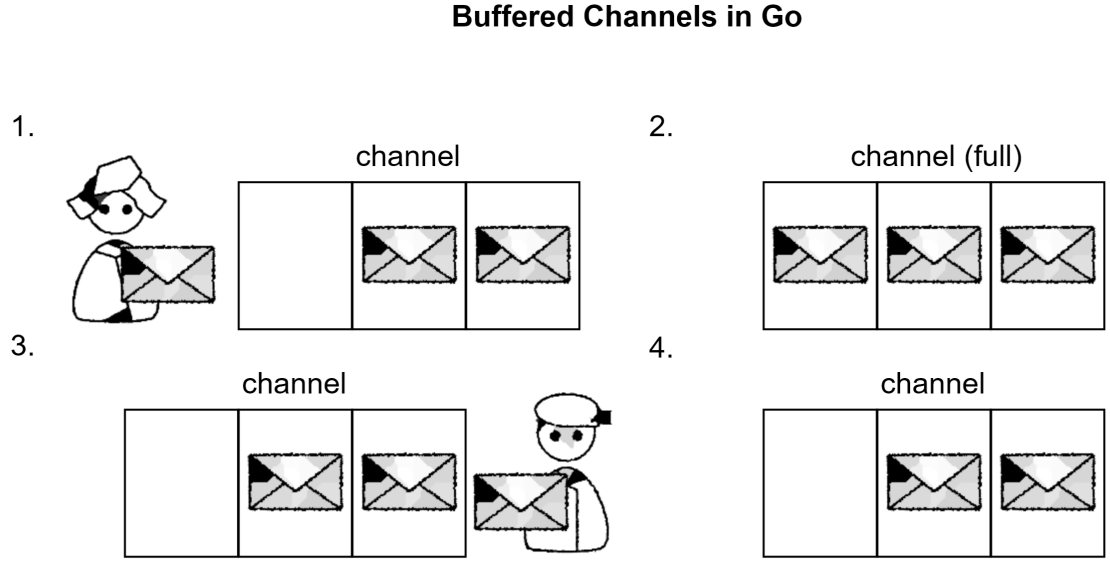

Figure 1.9: Buffered channel allowing for asynchronous communication.

(1) Here Alice (left) fills the channel from left to right with messages. (2) The channel is full and Alice is free to continue her operations. (3) Bob (right) takes the rightmost message from the channel. (4) Bob is free to continue his operations, leaving the channel.

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.