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

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

— 1 —

Introduction

1.1 High-level Computer Architecture Overview

1.1.1 System Review

To understand distributed systems, we must first review the architecture of a single computer.

Definition 1.1: Turing Machine

Conceptualized by Alan Turing in 1936, a Turing machine is a mathematical model of computation that defines an abstract machine that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, the machine can simulate the logic of any computer algorithm.

Definition 1.2: Vaun Neumann Architecture

The Von Neumann architecture, also known as the Princeton architecture, is a design architecture for an electronic digital computer with these components:

- **A processing unit** that contains an arithmetic logic unit and a control unit.
- **A memory unit** that stores data and instructions.
- **Input and output mechanisms**.

Fast forward, modern computers have the following components:

Definition 1.3: Modern Computer Components

- **CPU:** Central Processing Unit. The brain of the computer that performs instructions.
- **Memory:** Stores data and instructions.
- **Storage:** Hard drives, SSDs, etc.
- **Network Interface:** Connects the computer to the network.
- **Input/Output Devices (I/O):** Keyboard, mouse, monitor, etc.
- **Motherboard:** The central printed circuit board that interconnects all of the computer's components, including the CPU, storage devices, and I/O devices.

Before diving deeper into the inner workings of a single computer, let's define a distrusted system:

Definition 1.4: Distributed System

A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. The components interact with one another in order to achieve a common goal.

In the words of Andrew S. Tanenbaum,

> *"A set of nodes, connected by a network, which appear to its users as a single coherent system."*

or in the words of Leslie Lamport,

> *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*

Tip: **Andrew S. Tanenbaum** is a computer scientist and professor emeritus at the Vrije Universiteit Amsterdam in the Netherlands who is best known for his books on computer science. **Leslie Lamport** is an American computer scientist known for his work in distributed systems and as the initial developer of the document preparation system LaTeX.

1.1.2 CPU and Memory Orchestration Review

Now at a high-lever, we discuss how the a system interacts with all its components to perform tasks.

Definition 1.5: CPU (Central Processing Unit)

The CPU is made of the following components:

- **ALU (Arithmetic Logic Unit):** Performs arithmetic and logical operations.
- **Control Unit:** Manages the execution of instructions.
- **Registers:** Small, fast storage locations in the CPU that temporarily hold data and instructions.

Definition 1.6: Memory Segments

A program's memory is typically divided into several segments:

- **Text Segment:** Contains the executable code.
- **Data Segment:** Stores global and static variables.
- **System Stack:** A memory region that manages temporary data related to function calls in a first-in-last-out manner.
- **System Heap:** A memory region that dynamically allocates references to data from the stack memory.

Definition 1.7: Instruction Execution Cycle

The instruction execution cycle, also known as the *fetch-decode-execute* cycle, is the process by which the CPU processes instructions. In each cycle:

1. **Fetch:** The CPU retrieves an instruction from memory using the *instruction pointer* (or program counter).
2. **Decode:** The instruction is interpreted to determine what action is required.
3. **Execute:** The CPU performs the instruction's operation, which may involve arithmetic calculations, memory accesses, or I/O operations.

The CPU performs instructions via the following steps:

Definition 1.8: CPU Registers

Registers are small, high-speed storage locations within the CPU that temporarily hold data, instructions, and control information. Key registers include:

- **Instruction Pointer (Program Counter):** Holds **addresses**, which are the locations of the next instruction to fetch.
- **Stack Pointer:** Points to the top of the current stack in memory.
- **General-Purpose Registers:** Used for arithmetic operations and temporary data storage.

Definition 1.9: RAM and Volatile Memory

RAM (Random Access Memory) is a type of volatile memory used to store data and instructions that are actively used by the CPU. Since it is volatile, its contents are lost when the computer is powered off.

Definition 1.10: Physical Storage and I/O Devices

Physical storage refers to non-volatile memory devices such as hard drives and SSDs, which retain data without power. Many of these devices are accessed via input/output (I/O) operations and are thus considered part of the system's I/O mechanism.

Definition 1.11: Virtual Memory and Address Translation

Virtual memory is a memory management technique that provides an abstraction of a large, contiguous memory space. It works by mapping virtual addresses used by programs to physical addresses in RAM via structures such as page tables, which are managed by the Memory Management Unit (MMU).

Definition 1.12: CPU Cores

A CPU core is a physical processing unit within a central processing unit (CPU) responsible for executing instructions and performing computations. Modern CPUs often contain multiple cores, enabling them to handle multiple tasks at the same time.

Definition 1.13: Task, Job, and Process

- A **Task** is a single unit of work in various states (waiting, running, completed).
- A **Job** is a high-level operation comprising multiple tasks.
- A **Process** is an executing instance of a program that manages system resources instructing the CPU to execute tasks.

Definition 1.14: Threads: Concurrency & Parallelism

A **thread** is a unit of logic (a segment of code) to be executed by a CPU core. **A core can only run one thread at a time**. The core itself is called the **hardware-thread**, while our units of logic are called **software-threads** or **OS-threads**.

The OS system scheduler manages the hardware-threads, and assigns software-threads to them. Switching between software-threads on a hardware-thread is called **context switching**. Context switching is expensive, as it requires saving the current state of the software-thread and loading the state of the new software-thread. Though it provides the illusion of tasks running simultaneously, called **concurrency**.

With multiple cores come **multi-threading**, where multiple threads run on other cores simultaneously. This true simultaneity is called **parallelism**.

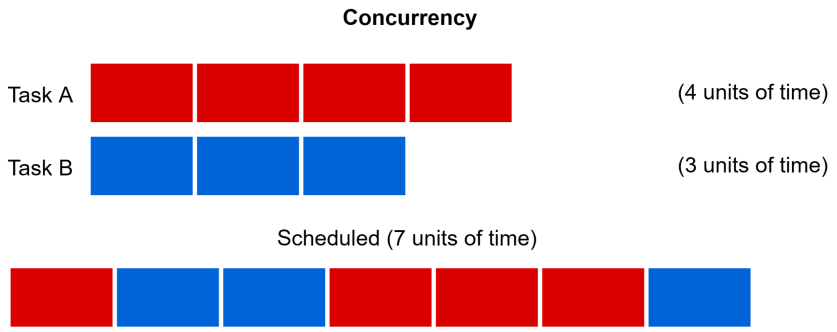


Figure 1.1: Concurrency: Multiple software-threads running on a single hardware-thread.

In reality, many tasks preform I/O operations, which don't concern the CPU:

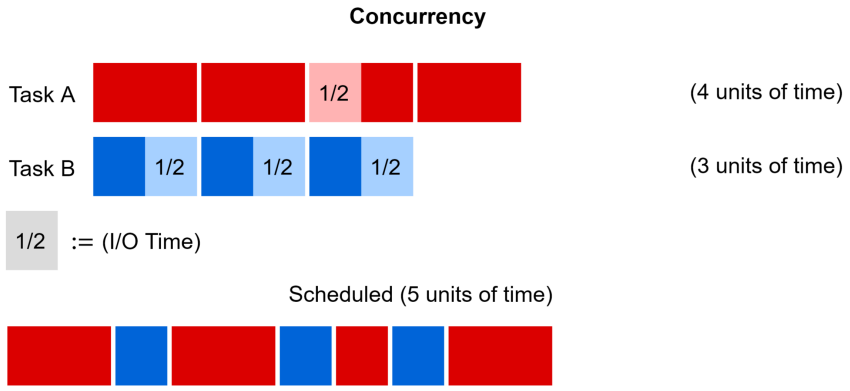


Figure 1.2: Concurrency with I/O: Multiple software-threads running on a single hardware-thread.

Now since the CPU isn't idle on I/O operations, the overall time between tasks is cut significantly.

Definition 1.15: Kernel

The kernel is the central component of the operating system. It manages hardware resources—including the CPU, memory, and I/O devices—and provides core services such as process management, memory management, and device control.

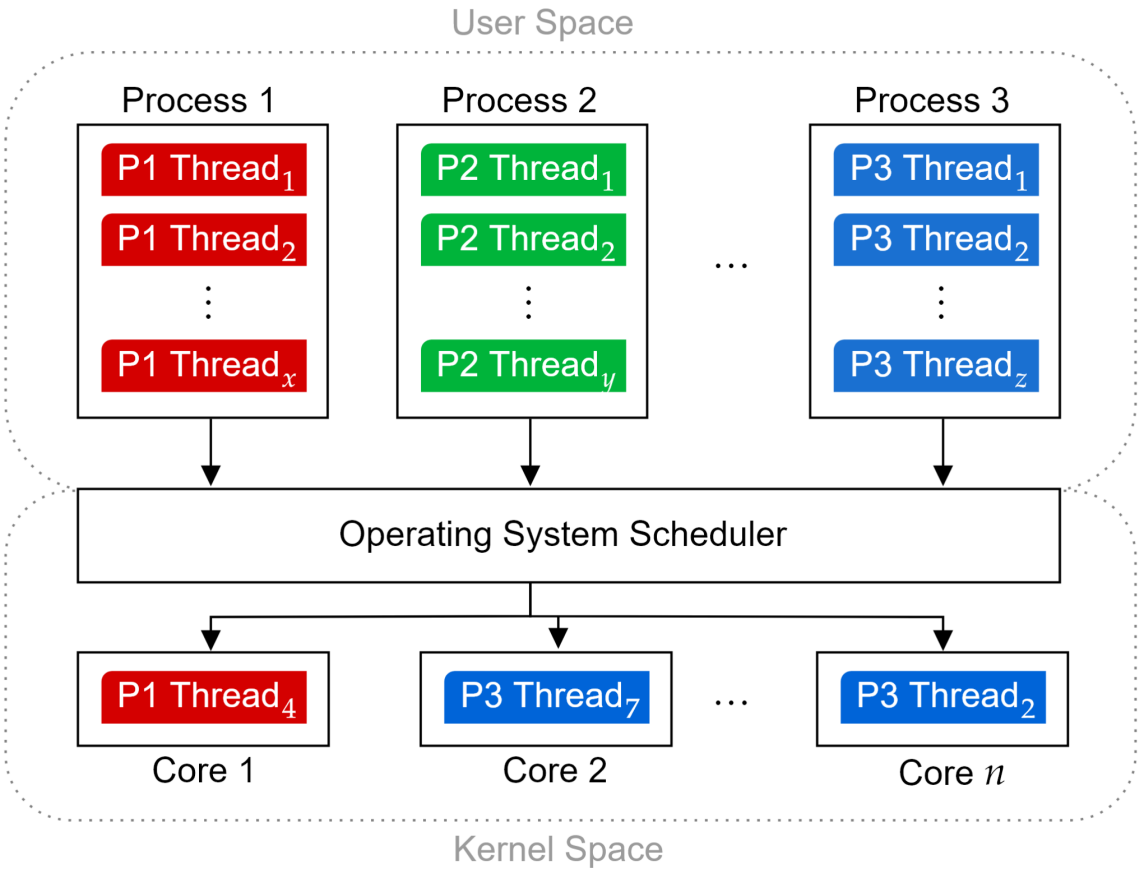


Figure 1.3: Process threads scheduled by the OS kernel and processed by the CPU.

In summary, what we need to know are these key points:

- The CPU executes instructions via processing units called cores/hardware-threads.
- The OS schedules software-threads from processes to run on hardware-threads.
- A core can only run one software-thread at a time, but can switch between them.
- Context switching on a single core is called concurrency, while utilizing multiple cores in unison (multi-threading) is called parallelism.

1.1.3 Motivation for Distributed Systems

Distributed systems cover a vast and diverse range of applications, including:

- **Offloading Computation:** Perhaps a system A offloads a heavy computation to system B.
- **Fault Tolerance:** If a critical system A fails, an almost identical system B can take over.
- **Load Balancing:** Say a system A is overwhelmed with requests, it can distribute the load to system B, acting as one system, from the requests point of view.

In todays market there are numerous applications of distributed systems, such as: Cloud Computing, Social Networks, E-commerce, Streaming Services, Search Engines, Renting Computation (AI training), etc.

Let's begin to define the problem space. Say there be two individuals Alice and Bob, who wish to communicate:

Figure 1.4: Alice sends a letter m_1 overseas to Bob.

How does Alice know that her message m_1 was received by Bob? Bob would have to send a message back to Alice, acknowledging the receipt of m_1.

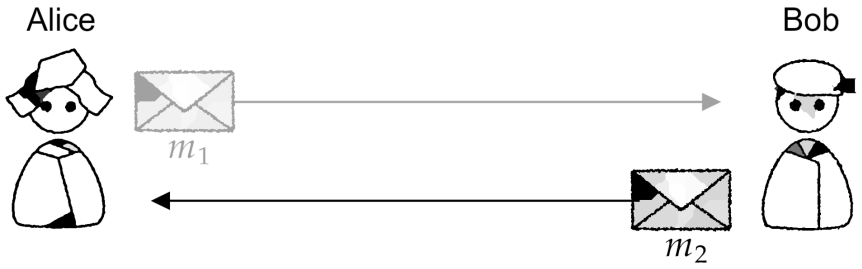


Figure 1.5: Bob sends an acknowledgment letter back to Alice.

Though problems can arise, what if Alice's letter gets lost in the mail, what if Bob receives multiple letters from Alice, how does Bob know which letter is the most recent? These are the fundamental problems of distributed systems.

1.2 Understanding RPCs & Synchronization with Go

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.

1.2.1 Establishing a Client-Server Connection

Definition 2.1: 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 2.2: 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 2.3: 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:

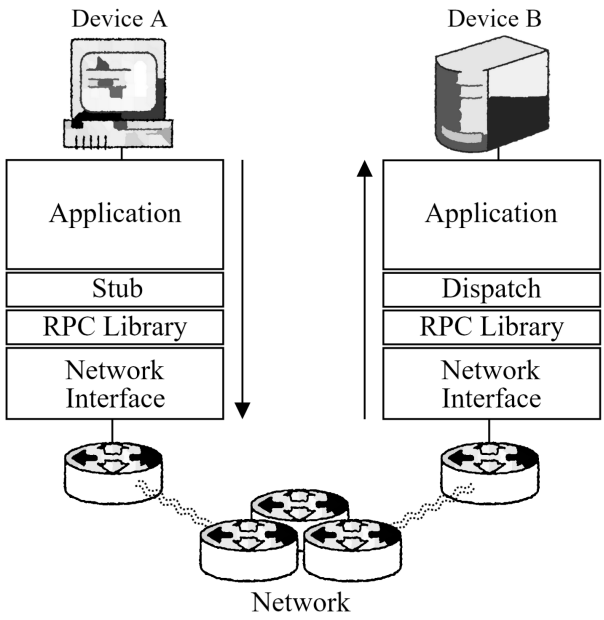


Figure 1.6: Client system A making a request to Server system B over RPC.

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

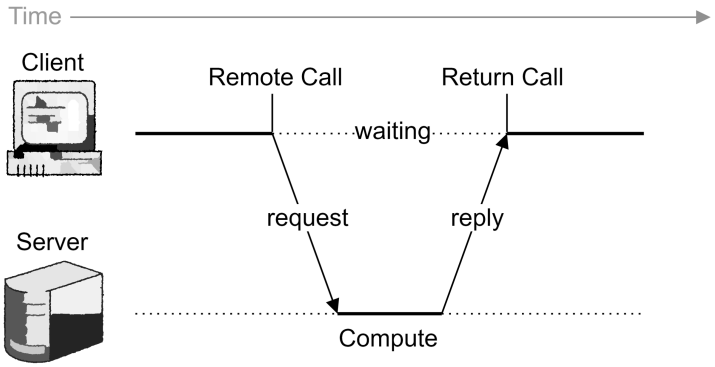


Figure 1.7: RPC call stack over time.

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 2.4: 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:

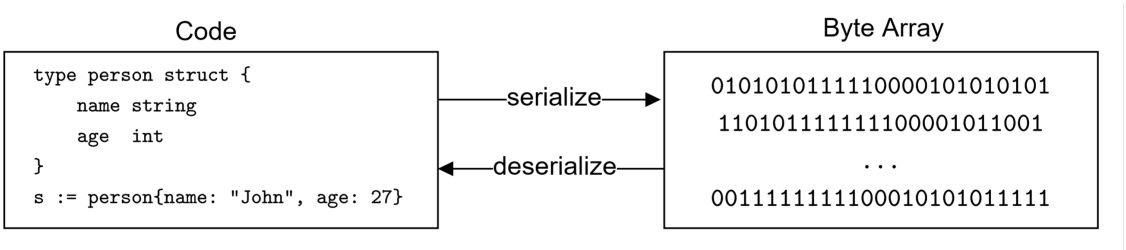


Figure 1.8: Serialization and Deserialization of data.

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

Theorem 2.1: 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 2.5: 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 2.6: 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 2.7: 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 2.8: 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:

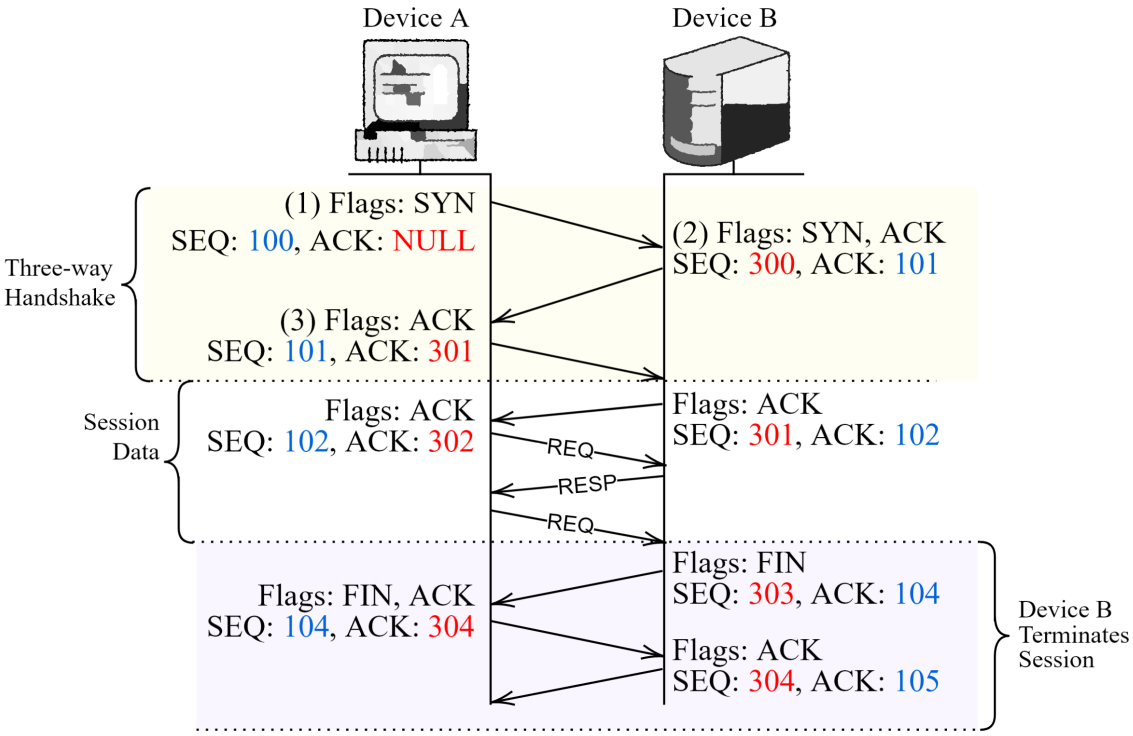


Figure 1.9: TCP Handshake, data transfer, and session termination.

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

1.2.2 Asynchronous Function Calls

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

Definition 2.9: 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.

To illustrate the difference between synchronous and asynchronous function calls, consider the following diagram:

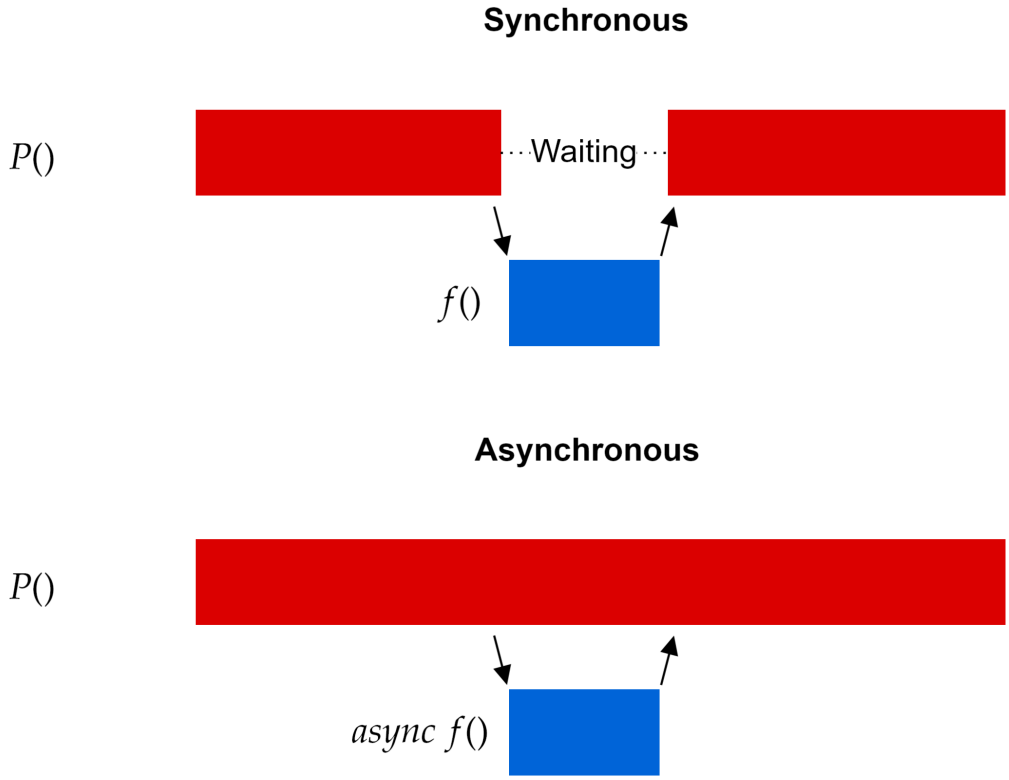


Figure 1.10: Synchronous vs. Asynchronous Function Calls.

- Function $P()$ (above) is the main function for which our program is running. It makes a synchronous call to function $f()$, which blocks the main program flow until $f()$ completes.
- In contrast, function $P()$ (below) instead makes an asynchronous call to function $f()$, allowing the main program to continue executing while $f()$ runs independently.

Definition 2.10: 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.

The below details the Go runtime scheduler. **Note**: that overtime the Go runtime's algorithm may change to improve performance. This isn't key to understanding the content of this text, but is provided for completeness sake. Here—at the time of writing—is how it works at a high-level:

Definition 2.11: 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).

The next page includes a diagram of the Go runtime scheduler above.

To illustrate the Go runtime scheduler on a high-level, consider the following diagram in contrast to Figure 1.3:

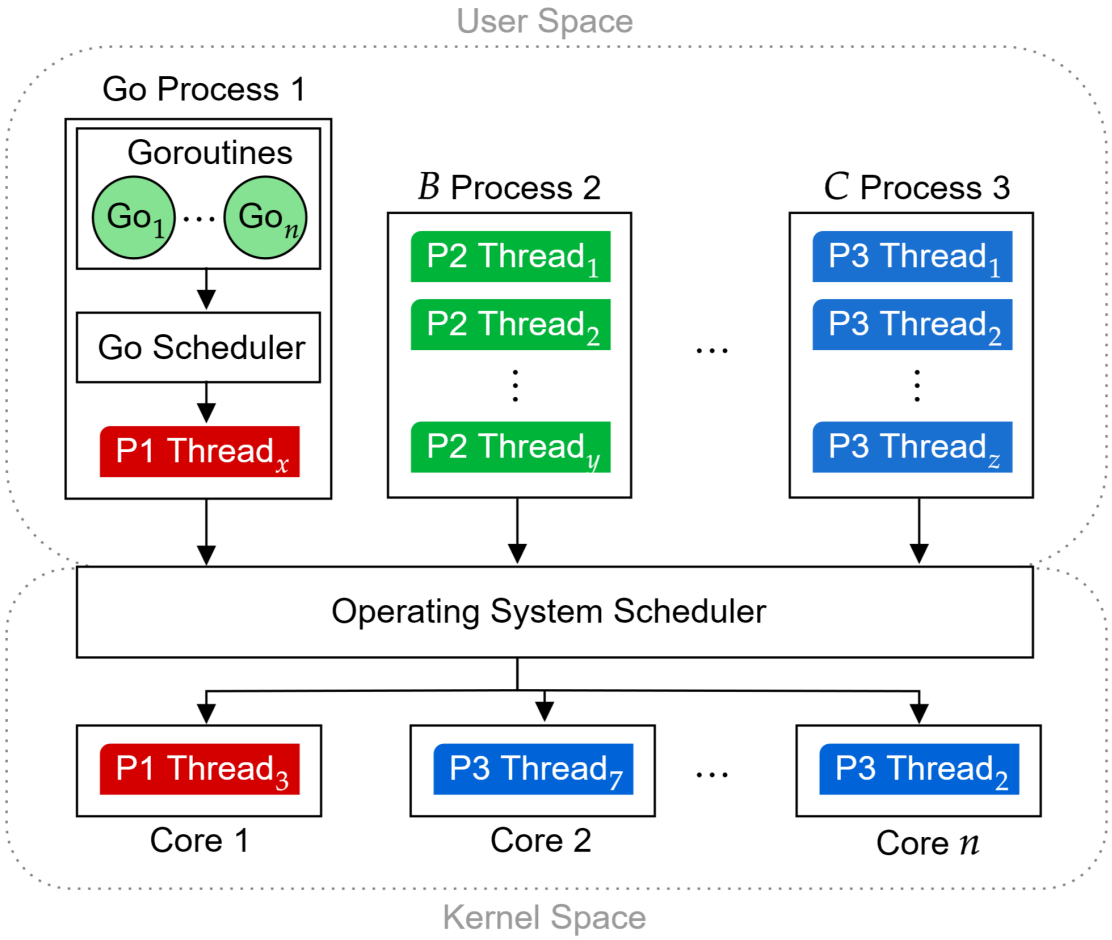


Figure 1.11: Go runtime scheduler with G, M, P entities.

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:

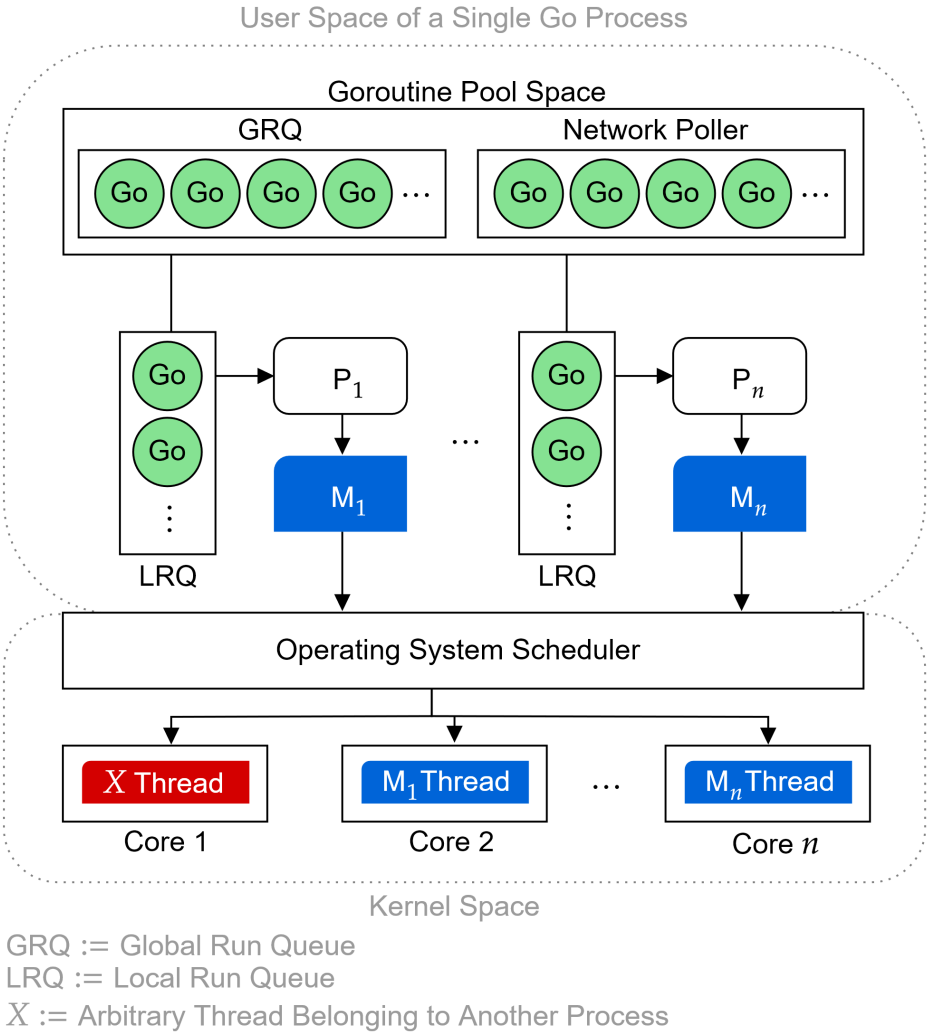


Figure 1.12: Go runtime scheduler within a single process.

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 2.2: 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 2.1: 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 2.3: 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 2.2: Count to n using Goroutines Corollary

Continuing off from the previous example (2.1), 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 2.4: 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 2.12: 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.

1.2.3 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 2.13: 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 2.14: 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 2.15: 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 2.3: 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 (2.3) using a mutex:

Definition 2.16: 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 2.17: 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 2.4: 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. ■

1.2.4 References & Pointers in Go

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

Definition 2.18: 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 2.19: 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 2.5: 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 2.20: 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
```

1.2.5 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 2.21: 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 2.6: 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 2.22: 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 2.7: Deferred Function Calls

Consider the previous example (2.6) 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. ■

1.2.6 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 2.23: 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. Channels must be closed after use to prevent memory leaks with `close()`.

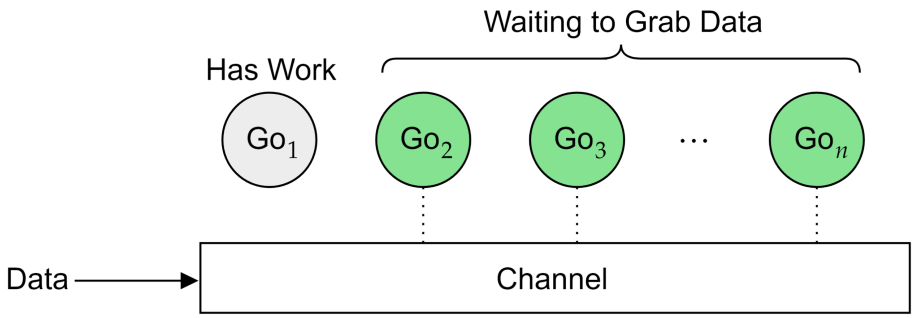


Figure 1.13: A collection of Goroutines competing for the next channel resource.

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

Example 2.8: Synchronizing Incoming Data Processing with Channels

Consider the following example of downloading and processing data concurrently:

Listing 1.8: Using Channels to Synchronize Downloading and Processing

```
package main
import (
    "fmt"
    "sync"
    "time"
)

// Download function simulates downloading data and sends a signal when
    done
func download(i int, ch chan int) {
    fmt.Printf("Downloading: Resource_%d...\n", i)
    time.Sleep(5 * time.Second) // Simulating download time
    fmt.Printf("Download complete: Resource_%d\n", i)
    ch <- i // Send signal that download is complete
}

// Process function waits for a signal before processing
func process(ch chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    i := <-ch // Wait for download to complete
    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() {
    n := 5
    ch := make(chan int) // Unbuffered channel
    var wg sync.WaitGroup
    wg.Add(n)
    // Spawn n goroutines to download and process resources
        concurrently
    for i := 0; i < n; i++ {
        go download(i, ch)
        go process(ch, &wg)
    }

    wg.Wait()
    close(ch) // Close channel after all downloads are completed
    fmt.Println("Main function exits")
}
```

■

Theorem 2.5: 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:

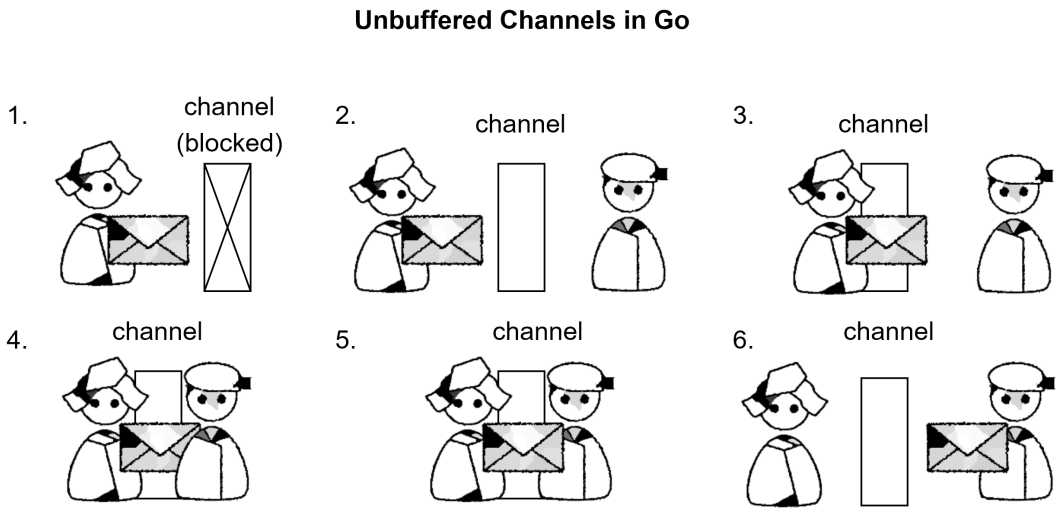


Figure 1.14: Handshake process of an unbuffered channel.

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.

Theorem 2.6: Unbeffered Blocking

Let A and B be two goroutines attempting to send messages over an unbuffered channel. If A enters the channel first, B will be blocked until A finishes the transaction.

The previous example (2.8) uses an unbuffered channel.

In contrast buffered channels allow for a more asynchronous approach:

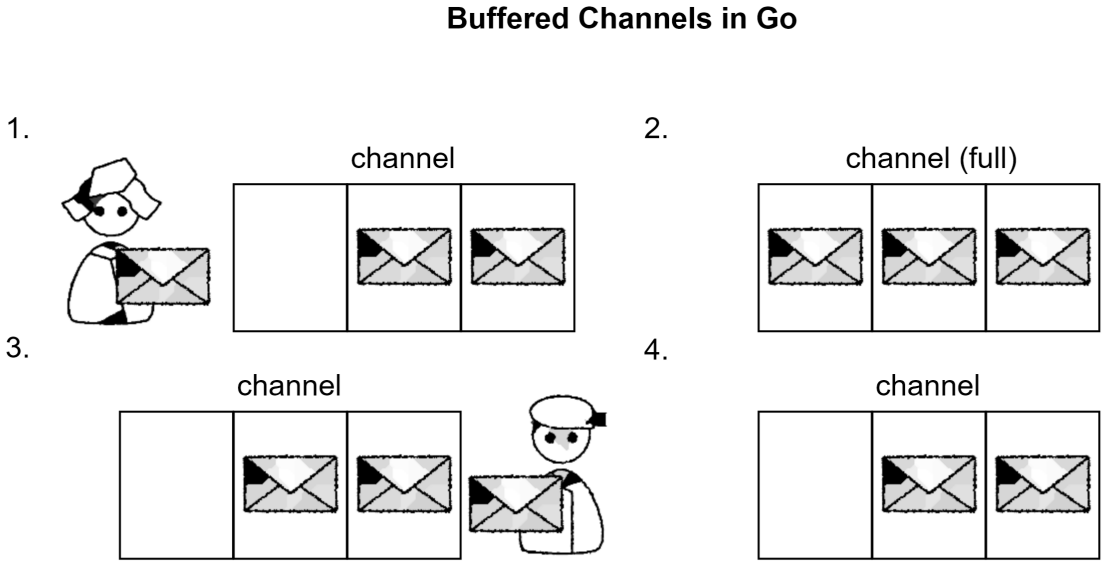


Figure 1.15: Buffered channel allowing for asynchronous communication.

(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.

To make the last example (2.8) use a buffered channel, we can modify the channel creation.

Example 2.9: Using Buffered Channels

Consider the previous example (2.8) with a buffered channel:

Listing 1.9: Using Buffered Channels to Synchronize Downloading and Processing

```
...
ch := make(chan int, n) // Buffered channel with a capacity of 5
...
```

Here, the channel has a buffer size of n, allowing up to n messages to be stored before the receiver is ready. ■

1.2.7 Task, Data, and Pipeline Parallelism

Task, data, and pipeline parallelism are three common forms of parallelism:

Definition 2.24: Task Parallelism

Task parallelism involves running multiple tasks simultaneously. Each task is independent and can run in parallel with other tasks, perhaps even on the same data.

In essence, we may think same data, different tasks:

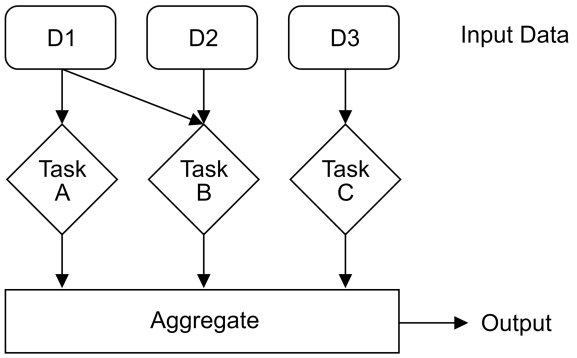


Figure 1.16: Task Parallelism Culminating into an Aggregate Result

Definition 2.25: Data Parallelism

Data parallelism involves running the same task on multiple data items. Each task is identical, but the data is different.

We may think same task, different data:

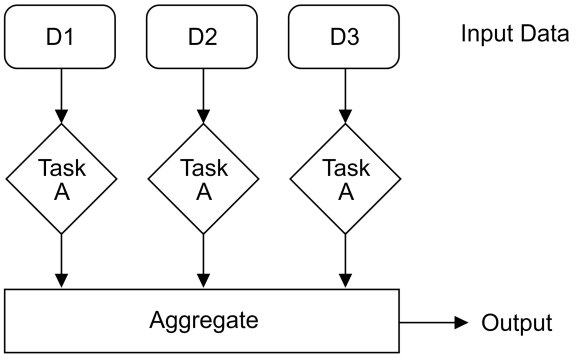


Figure 1.17: Data Parallelism Culminating into an Aggregate Result

To continue, we have:

Definition 2.26: Pipeline Parallelism

Pipeline parallelism involves breaking a task into multiple stages, each of which can be executed concurrently. The output of one stage is the input to the next stage.

For instance, consider the following pipeline:

- **Task A**: "Search for a flight." (1 time unit)
- **Task B**: "Book a flight." (1 time unit)

First consider the scenario where we only have one resource to work with, resulting in concurrency:

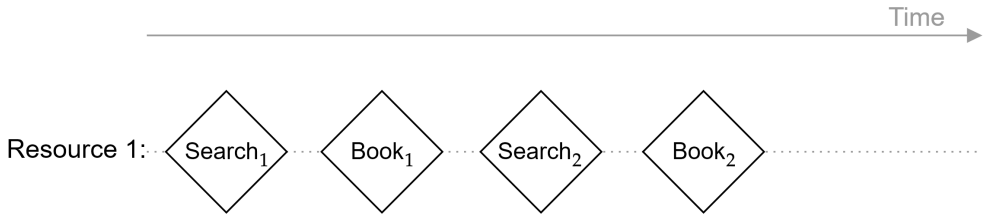


Figure 1.18: Searching and Booking flights concurrently

Now consider the scenario where we have two resources to work with, resulting in parallelism: In

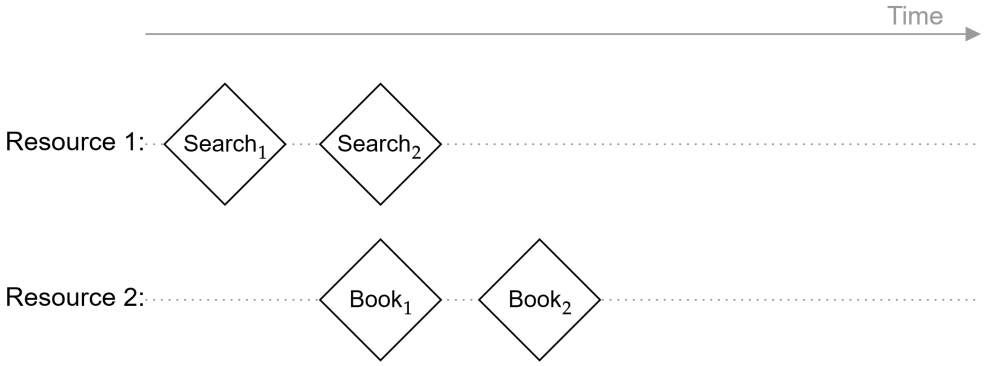


Figure 1.19: Searching and Booking flights in parallel

this case, once the first search is done, we can start booking the flight and search for the next flight in parallel.

1.2.8 Arrays & Slices in Go

Arrays in Go act like arrays in other languages, a fixed-size collection of items of the same type. Slices, on the other hand, allow us to work with a dynamically-sized sequence of elements.

Definition 2.27: Arrays in Go

An **array** in Go is a fixed-size collection of elements of the same data type. Arrays in Go are value types, meaning they are copied when assigned to a new variable.

Arrays are declared using the syntax:

```
var arr [size]Type
```

For example, an array of integers with 5 elements:

```
var numbers [5]int
```

Elements in an array can be accessed using zero-based indexing:

```
numbers[0] = 10  // Assign value
fmt.Println(numbers[0]) // Access value
```

Arrays cannot be resized, and their size must be known at compile time. For dynamic collections, slices are preferred.

Example 2.10: Doubling Items in an array

Consider the following example where we double each element in an array:

```
package main

import "fmt"

func main() {
    // Initialize an array
    numbers := [5]int{1, 2, 3, 4, 5}

    // Double each element in the array
    for i := 0; i < len(numbers); i++ {
        numbers[i] *= 2 // Shorthand for numbers[i] = numbers[i] * 2
    }

    // Print the modified array
    fmt.Println(numbers)
}
// Output: [2 4 6 8 10]
```

■

In contrast, slices:

Definition 2.28: Slices in Go

A **slice** is a dynamically-sized reference to a portion of a single underlying array.
Slices are declared using square brackets without specifying a fixed size:

```
var numbers []int // A slice of integers
```

Slices are typically created using the `make` function or by slicing an existing array:

```
// Using make()
numbers := make([]int, 5) // Creates a slice with length 5

// Slicing an array
arr := [5]int{1, 2, 3, 4, 5}
slice := arr[1:4] // Slice from index 1 to 3 -> {2, 3, 4}
```

Slices maintain a reference to the original array, meaning modifications affect both:

```
arr := [5]int{1, 2, 3, 4, 5}
slice := arr[1:3]
slice[0] = 99 // Modifies arr[1] as well
fmt.Println(arr)  // Output: [1 99 3 4 5]
```

The `append()` function modifies slices given there's enough capacity:

```
arr := [5]int{1, 2, 3, 4, 5}
slice := arr[:1] // Slice from index 0 to 0 -> {1}
slice = append(slice, 7, 8) // Adds elements to the slice
fmt.Println(slice) // Output: [1 7 8]
fmt.Println(arr)   // Output: [1 7 8 4 5] (modified)
```

If `append()` exceeds the slice's capacity, a new array is allocated and referenced by the slice:

```
... // Previous code
fmt.Println(cap(slice)) // Output: 5 (capacity of the slice)
slice = append(slice, 7, 8, 9, 10, 11) // Exceeds capacity, (6 total)
fmt.Println(slice) // Output: [1 7 8 9 10 11]
fmt.Println(arr)   // Output: [1 2 3 4 5] (unchanged)
```

To copy an array to a slice, use the `copy()` function:

```
arr := [5]int{1, 2, 3, 4, 5}
slice := make([]int, len(arr))
copy(slice, arr) // Syntax: copy(destination, source)
slice[0] = 99
fmt.Println(slice) // Output: [99 2 3 4 5]
fmt.Println(arr)   // Output: [1 2 3 4 5] (unchanged)
```

1.2.9 Repeating Tasks: Tick and Ticker in Go

In Go, the `time` package provides two types for repeating tasks at regular intervals:

Definition 2.29: `time.Tick` and `time.Ticker` in Go

The `time` package in Go provides two mechanisms for scheduling repeated tasks at fixed intervals:

- `time.Tick(duration)`: Returns a channel that sends the current time at regular intervals. It is a convenience function but cannot be stopped.
- `time.NewTicker(duration)`: Creates a `Ticker` object, which provides a `.Stop()` method to halt the ticker. Additionally the `.C` returns a channel from which the signal can be read. This channel is read only, hence a type of `<-chan time.Time`.

Example 2.11: Record a signal n times every second

Say we want to record a signal n times every second. We can use a `time.Ticker`:

```
package main
import "fmt"; import "time"; import "sync"

func Status(ch <-chan time.Time, wg *sync.WaitGroup) {
    defer wg.Done()
    <-ch // Wait for signal
    fmt.Println("Status: OK")
}

func main() {

    n := 5
    ticker := time.NewTicker(time.Second)
    tickerChan := ticker.C
    var wg sync.WaitGroup

    // Spawns n goroutines which waiting for the signal
    for i := 0; i < n; i++ {
        wg.Add(1)
        go Status(tickerChan, &wg)
    }
    wg.Wait()
    ticker.Stop()
}
```

This type of example can be extended to perform any task at a given interval. ■

1.2.10 Conditionally Reading from Channels: Select in Go

Now to discuss conditionally reading from multiple channels:

Definition 2.30: `select` in Go

The `select` statement in Go allows a goroutine to wait on multiple channels and perform an action as soon as one of them receives a value:

```
select {
case val := <-ch1:
    // Received from ch1
case val := <-ch2:
    // Received from ch2
default:
    // Executes if no channels are ready
}
```

Note: `default` is optional. If multiple channels are ready, **one is chosen at random.**

Example 2.12: Conditianlly Waiting for a Signal

Consider a situation where we conditionally wait on a channel for a signal:

```
... // Imported "math/rand"
func Status(rc chan string) {
    for { // Loops forever sending a message every second
        // Randomly select a status
        rc <- []string{"Great", "Ok", "Slow"}[rand.Intn(3)]
        time.Sleep(1 * time.Second)
    }
}

func main() {
    replyChannel := make(chan string) // Channel for receiving status
    go Status(replyChannel) // Asynchronous status generator
    for {  // Loop forever waiting for such status
        select {
        case <-replyChannel:
            fmt.Println("Status:", <-replyChannel)
        default:
            fmt.Println("Waiting...")
            time.Sleep(351 * time.Millisecond)
        }
    }
}
```

■

1.3 Time, Clocks, and Logical Ordering

1.3.1 Accuracy of Time: Atomic Clocks & NTP

Time allows us to order and identify events. Say we ran `time.Now()` on two different machines:

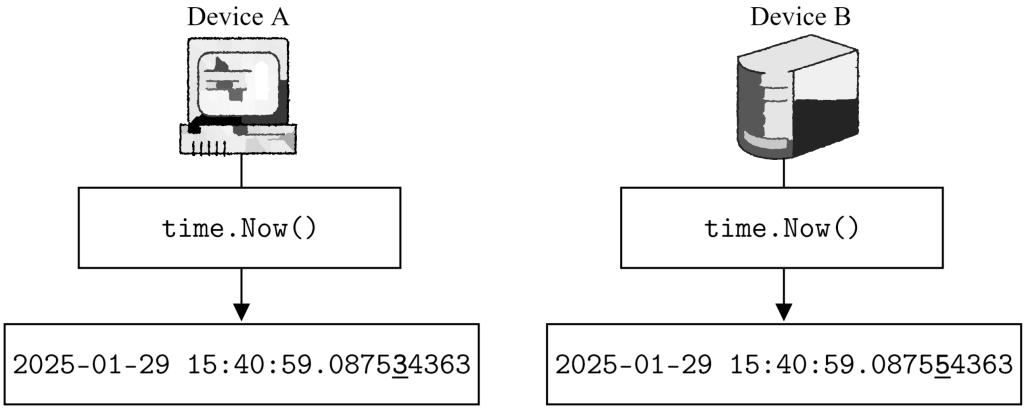


Figure 1.20: Using `time.Now()` on two different machines

Despite Device A appearing to be ahead of Device B, we cannot be certain via the following reasons:

Theorem 3.1: Clock Synchronization Impossibility

There are two key reasons why perfect clock synchronization is impossible:

- **Clock Skew:** There's a difference between every system clock (ideally 0), as they maintain their own local clock via a hardware oscillator incrementing a counter register.
- **Clock Drift:** Even if systems initialize with a reference time, their clocks will inevitably diverge due to variations in manufacturing, age, or environmental factors such as temperature. We measure the deviation by, $\frac{dC}{dt} = 1 + \rho$, where C is the clock time and t is the real time, and ρ (rho) is the drift rate (ideally 0).

We may formalize what we may consider synchronized clocks as follows:

Definition 3.1: Clock Synchronization Threshold

Let there be two clocks C_i and C_j. They are (delta) δ-synchronized if for all t time units:

$$|C_i(t) - C_j(t)| \leq \delta$$

E.g., C_i and C_j are δ-synchronized within 10ms if $|C_i(t) - C_j(t)| \leq 10ms$

In practice we to achieve semi-synchronized clocks, we developed the following protocol:

Definition 3.2: Network Time Protocol (NTP)

The NTP is a protocol synchronizes network clocks via a ground-truth time distribution system. The ground-truth time is are GPS satellite **atomic clocks**, which exhibit negligible drift over millions of years.

NTP employs a **round-trip time (RTT)** calculation to estimate the clock offset request latency. It also organizes synchronization strength into a **stratum hierarchy**, where lower-numbered stratums indicate more accurate time sources:

- **Stratum 0:** Ground truth **atomic clocks/GPS receivers**.
- **Stratum 1:** NTP servers that directly synchronize with Stratum 0 reference clocks.
- **Stratum 2:** NTP servers synchronized to Stratum 1 servers.
- **Stratum 3 and beyond:** Weaker NTP servers synchronized to higher-stratum servers.
- **Stratum 16:** A system considered *unsynchronized* (e.g., a freshly booted system).

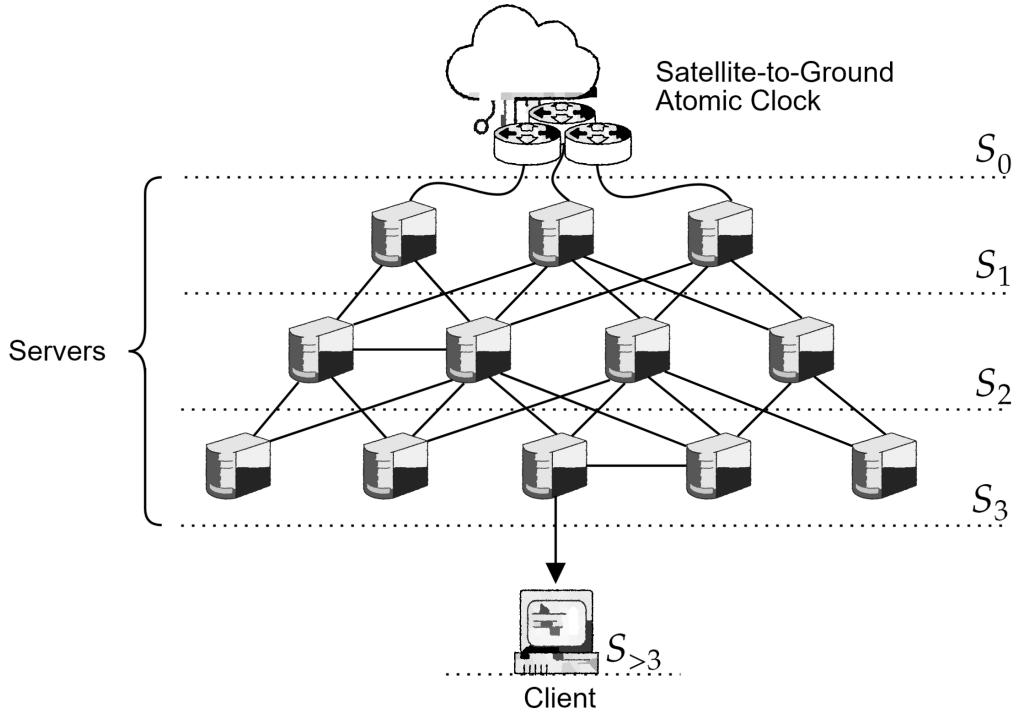


Figure 1.21: NTP Stratum Hierarchy From GPS Satellite Atomic Clock to Client.

1.3.2 Logical Clocks: Lamport & Vector Clocks

To get away from the limitations of physical clocks, we may use logical clocks to order events.

Definition 3.3: Logical Clocks

Let a and b be two events part of a totally ordered set of events. Let function $t(x)$ denote the time of event x. Then,

$$a \to b \implies t(a) < t(b)$$

Where $a \to b$ denotes that event a happens before b, which implies $t(a) < t(b)$.

We may become more formal about cause and effect relationships with the following definition:

Definition 3.4: Causal Order

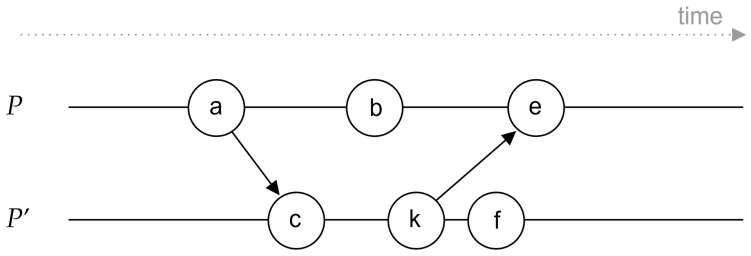
For r **execution trace** (sequence of events), the causal order relationship $\to_r$ is defined as:

- If a **happens before** b in the same process, then $a \to_r b$.
- If a is a **sender** and b the **receiver**, then $a \to_r b$.
- **Transitive Property**: if $a \to_r b$ and $b \to_r c$, then $a \to_r c$.
- Events a and b are **concurrent** (denoted as $a \parallel b$) if:
  - $a \not\to_r b$ and $b \not\to_r a$, meaning neither event happened before the other.

Example 3.1: Causal Order Example

Determine the causal order relationship between events in processes P and P':

- (a): $a \; ? \; b$; (b): $a \; ? \; k$; (c): $c \; ? \; b$; (d): $c \; ? \; e$



Answer on the next page. ■

Example (3.1) Answer: (a): $a \rightarrow_r b$; (b): $a \rightarrow_r k$; (c): $c \parallel b$; (d): $c \rightarrow_r e$.

Now we discuss a method that utilizes causal order, though assigns logical timestamps to events:

Definition 3.5: Lamport Clocks

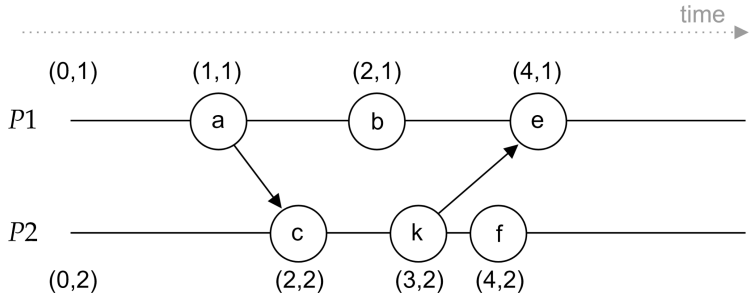
Named after Leslie Lamport, Lamport Clocks assign a logical timestamp to each event:

Let t_p store the logical time of process p. Then,

- **Initialization:** t_p is initialized to 0.
- **Timestamp Syntax:** Timestamps are tuples (t_p, p), assigned to each e event.
- **Incrementing:** For each e in process p, increment t_p by 1 and assign the (t_p, p) to e.
- **Sending:** If p sends a message m to process q, the timestamp included is $((t_p + 1), p)$.
- **Receiving:** Upon receiving message m, process q sets $t_q = \max((t_p + 1), t_q)$.

Example 3.2: Lamport Clocks Example (3.1) Extended

Consider the previous Example (3.1) with Lamport Clocks:



■

In practice the only thing we have access to are these logical timestamps, which we must evaluate:

Theorem 3.2: Comparing Lamport Timestamps Causalaity

Given two events a and b with timestamps $t(a)$ and $t(b)$, with r trace, we only guarantee:

- If $a \rightarrow_r b$, then $t(a) < t(b)$.
- If $t(a) \geq t(b)$, then $a \not\rightarrow_r b$.

We may now derive the following about concurrency:

Theorem 3.3: Non-causality

Two events a and b are concurrent ($a \parallel b$) under r trace if **both** conditions hold:

- $a \not\rightarrow_r b$ (a does not happen before b).
- $b \not\rightarrow_r a$ (b does not happen before a).

Lamport Clocks are useful for causal ordering, but they do not capture the full context of events:

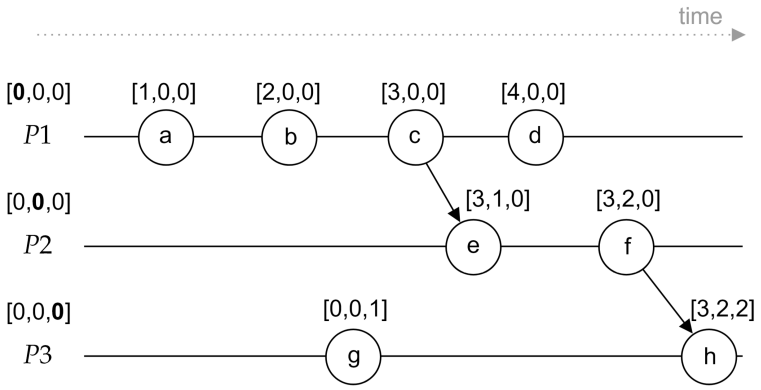
Definition 3.6: Vector Clocks

Let there be $p_1, p_2, \ldots, p_n$ processes each with a vector (array) v of size n. Index $v_i[i]$ stores the logical time of process p_i. Then, the following rules apply:

- **Initialization:** Each $\ell \in v_i$ of p_i is initialized to 0 (e.g., $[0, 0, \ldots, 0]$).
- **Incrementing:** For each event e in process p_i, increment $v_i[i]$ by 1.
- **Sending:** When p_i sends a message m to p_j, include v_i in m (**no increment**).
- **Receiving:** Upon receiving message m, process p_j sets $v_j[j] = \max(v_j[j], v_i[j])$.

Example 3.3: Vector Clocks Example

Observe the following events and their corresponding vector clocks:



■

There too is a method of comparing vector clocks:

Theorem 3.4: Comparing Vector Clocks

Given two vectors v_p and v_q for processes p and q, we may derive the following:

- $v_p \leq v_q \iff \forall \ell : v_p[\ell] \leq v_q[\ell]$
- $v_p < v_q \iff \forall \ell : v_p[\ell] < v_q[\ell]$
- $v_p <> v_q$ (non-comparable) $\iff \forall \ell : \neg(v_p < v_q) \wedge \neg(v_p > v_q)$

i.e.,

- $v_p \leq v_q$: if and only if all corresponding indexes in v_p are less than or equal to v_q.
- $v_p < v_q$: if and only if all corresponding indexes in v_p are less than v_q.
- $v_p <> v_q$: if and only if there are at least two element pairs between v_p and v_q that are greater and less than each other. (e.g., $v_p = [1, 2]$ and $v_q = [2, 1]$, as $1 < 2$ and $2 > 1$).

And for causality we have:

Theorem 3.5: Vector Clocks Causality

Given two events a and b with vector clocks $v(a)$ and $v(b)$, we may derive the following:

- If $v(a) < v(b)$, then $a \rightarrow_r b$.
- If $a \rightarrow_r b$, then $v(a) < v(b)$.

Exercise 3.1: Given the following Lamport timestamps, determine the causal order relationships:

(a) $(0, P)$? $(0, Q)$

(b) $(1, P)$? $(0, Q)$

(c) $(0, P)$? $(1, Q)$

Exercise 3.2: Given the following Vector clocks, determine the causal order relationships:

(a) $[0, 0]$? $[0, 0]$

(b) $[1, 0]$? $[0, 0]$

(c) $[0, 1]$? $[1, 0]$

Answers on the next page.

Answer 3.1: Given the following Lamport timestamps, determine the causal order relationships:

(a) $(0, P)$? $(0, Q)$: $P \parallel Q$

(b) $(1, P)$? $(0, Q)$: $P \parallel Q$

(c) $(0, P)$? $(1, Q)$: $Q \parallel P$

Note: The Lamport timestamps are not enough to determine causal order relationships between independent processes, unless we have the context of the execution trace.

Answer 3.2: Given the following Vector clocks, determine the causal order relationships:

(a) $[0, 0]$? $[0, 0]$: $[0, 0] \parallel [0, 0]$

(b) $[1, 0]$? $[0, 0]$: $[1, 0] \not\to [0, 0]$

(c) $[0, 1]$? $[1, 0]$: $[0, 1] <> [1, 0]$

Bibliography

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