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.

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

Task, Data, and Pipeline Parallelism

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

Definition 0.1: 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:

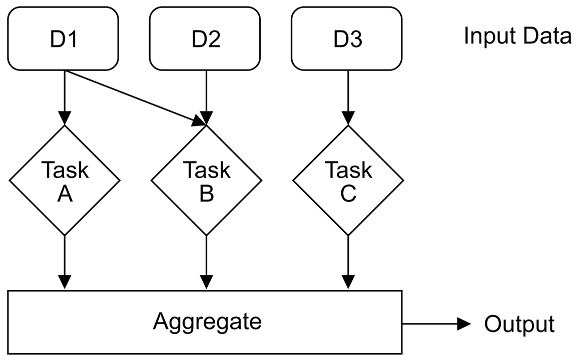

Figure 1.1: Task Parallelism Culminating into an Aggregate Result

Definition 0.2: 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:

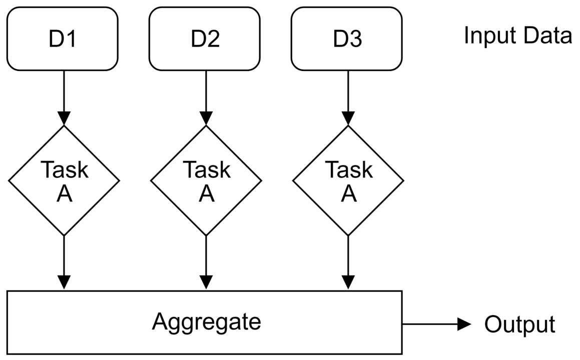

Figure 1.2: Data Parallelism Culminating into an Aggregate Result

To continue, we have:

Definition 0.3: 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:

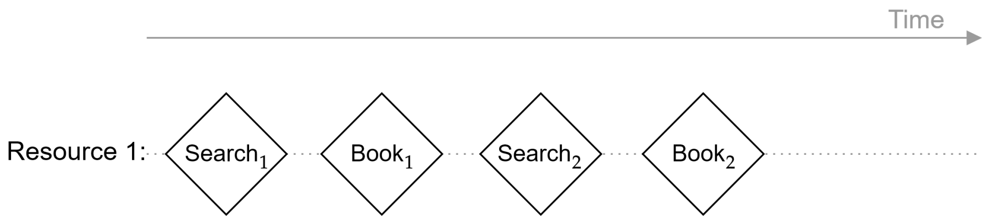

Figure 1.3: Searching and Booking flights concurrently

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

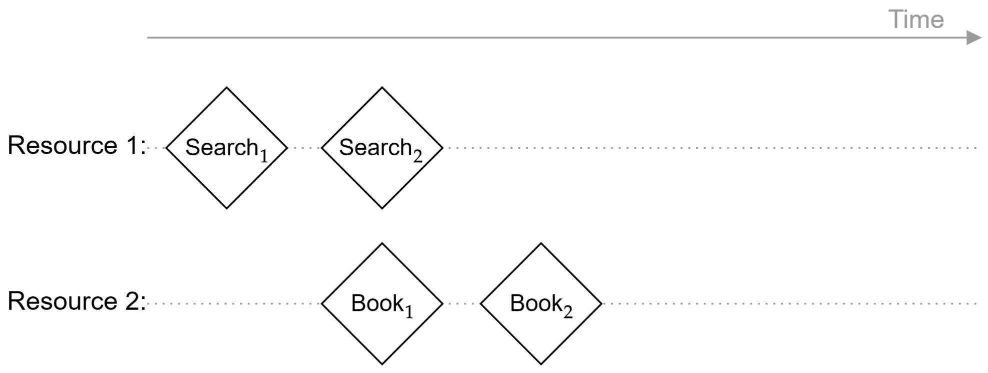

Figure 1.4: Searching and Booking flights in parallel

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

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 0.4: 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:

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

For example, an array of integers with 5 elements:

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

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

```go
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 0.1: Doubling Items in an array

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

```go
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 0.5: 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:

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

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

```go
// 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:

```go
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:

```go
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:

```go
... // 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:

```go
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)
```

Repeating Tasks: Tick and Ticker in Go

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

Definition 0.6: `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 0.2: Record a signal n times every second

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

```go
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. ■

Reading from Multiple Channels: Select in Go

Now to discuss conditionally reading from multiple channels:

Definition 0.7: 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:

```go
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 0.3: Conditianlly Waiting for a Signal

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

```go
... // 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)
        }
    }
}
```

■

Bibliography

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