Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Prof. Anna Arpaci-Dusseau**
and **Prof. Ioannis Liagouris** for teaching CS351: Distributed Systems
at Boston University [5].

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**.

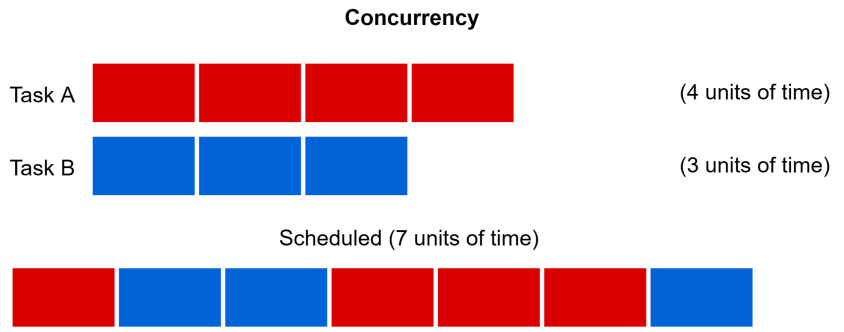


Figure 1.1: Concurrency: Multiple software-threads running on a single hardware-thread.

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

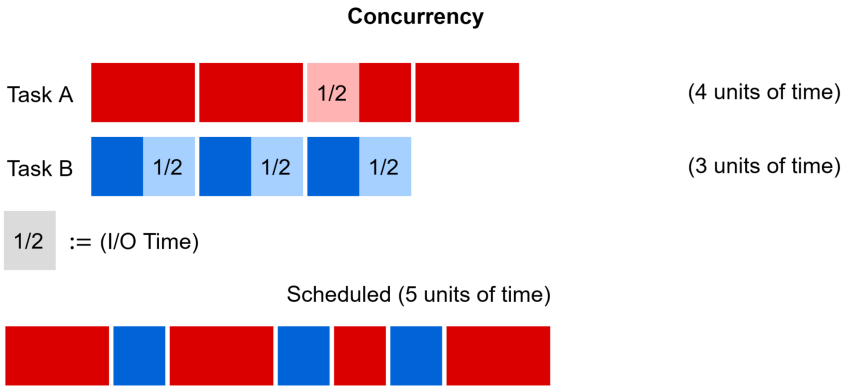


Figure 1.2: Concurrency with I/O: Multiple software-threads running on a single hardware-thread.

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.

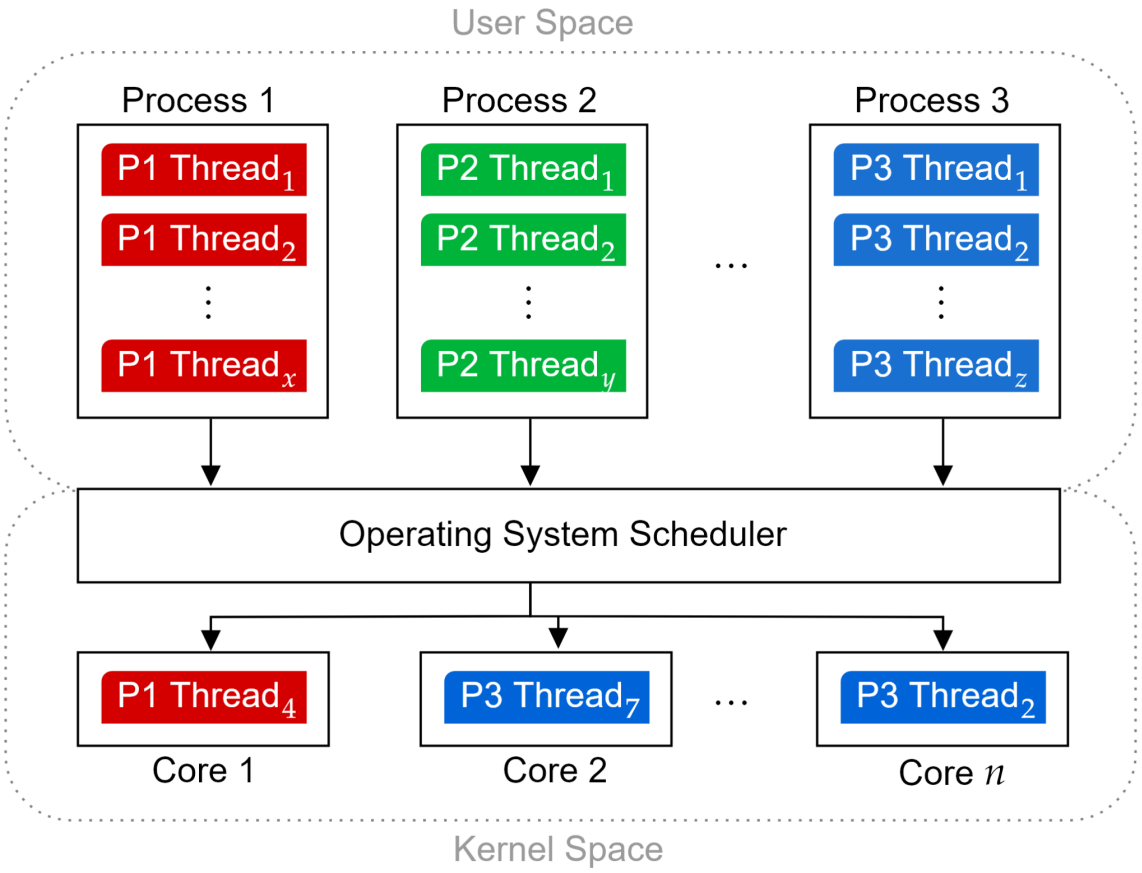


Figure 1.3: Process threads scheduled by the OS kernel and processed by the CPU.

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 are 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.

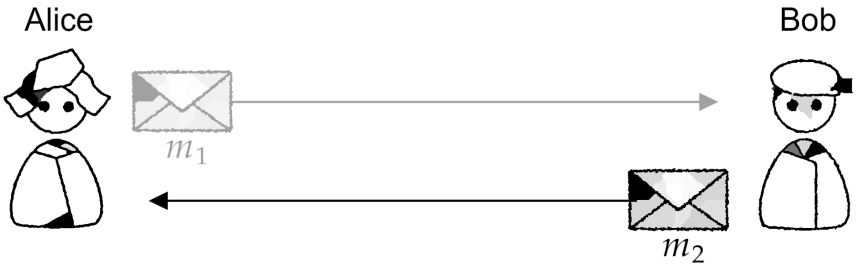


Figure 1.5: Bob sends an acknowledgment letter back to Alice.

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:

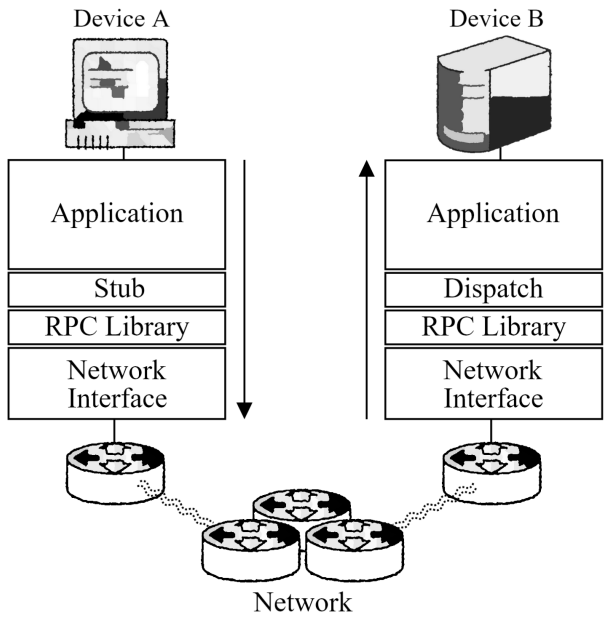


Figure 1.6: Client system A making a request to Server system B over RPC. This of which is reciprocated by B to return the result.

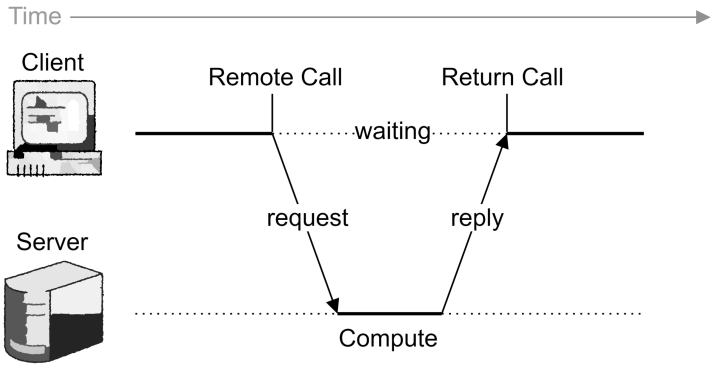


Figure 1.7: RPC call stack over time. Once the client makes the call it waits for the server to process the request and return the result. The programmer need not worry beyond sending the request and receiving the response. The RPC deals with all the heavy work of facilitating the communication.

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:

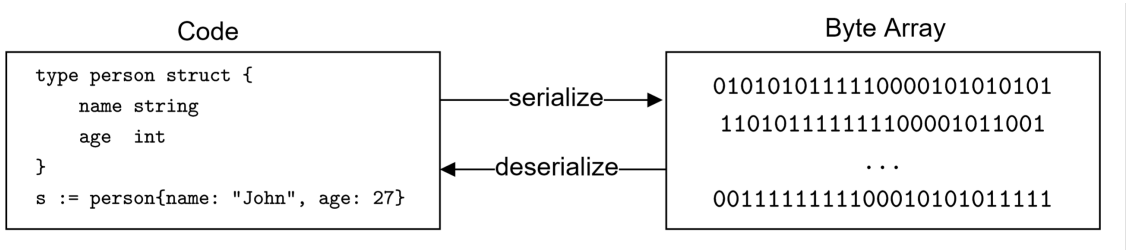


Figure 1.8: Serialization and Deserialization of data.

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 (client):** The client repeats requests until a response is heard. Reads alone are fine, but writes cause race conditions.
- **At-most-once (server):** The server handles duplicate requests from clients via some ID system, enabling cached responses (requires client to send IDs).

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 SYN, **FIN (finish)** is used.

Such a system preserves the order of packets sent and received, making it a **FIFO (First In First Out)** system.

We now combine at-least-once and at-most-once to create the following model:

Definition 2.8: Exactly-Once Model

Utilized both **at-least-once** and **at-most-once** (client and server) models. Hence, client send requests indefinitely and the server handles duplicates via using IDs provided by the client. Duplicate requests receive a response of the cached response.

Below we illustrate a simple TCP connection:

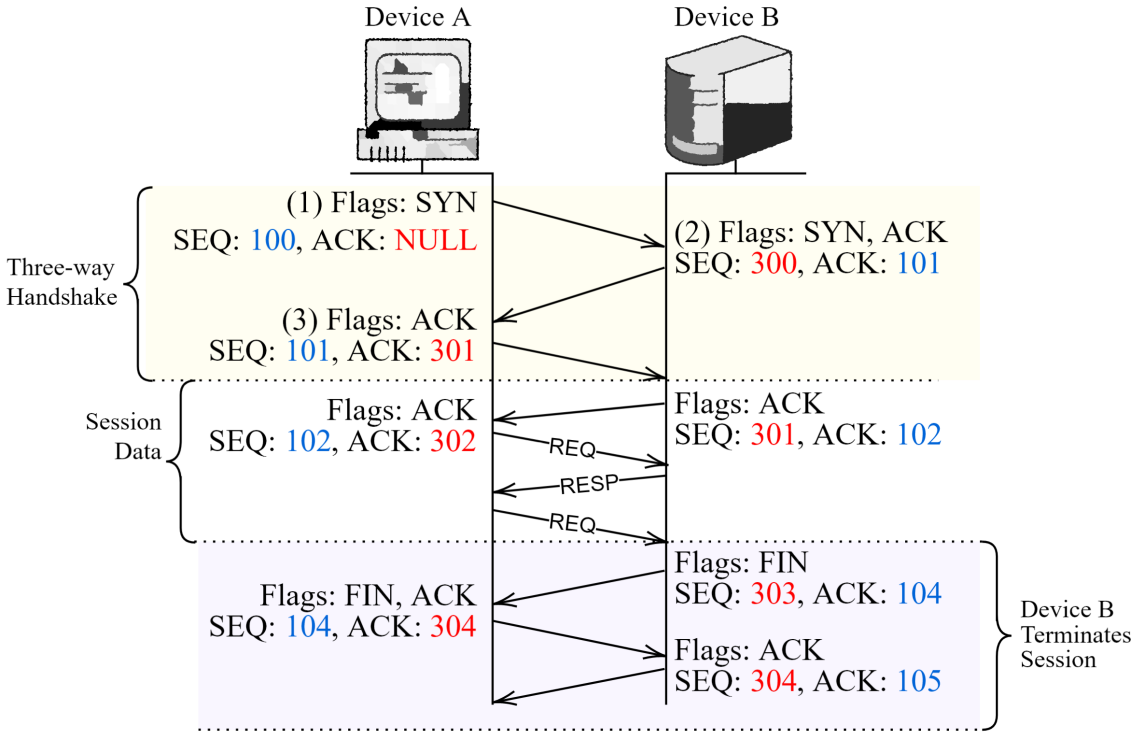


Figure 1.9: TCP Handshake, data transfer, and session termination. Here the client (Device A) begins a three-way handshake with the server (Device B) to establish a connection. Both start with arbitrary sequence numbers for security purposes. With each packet received the two devices increment their 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:

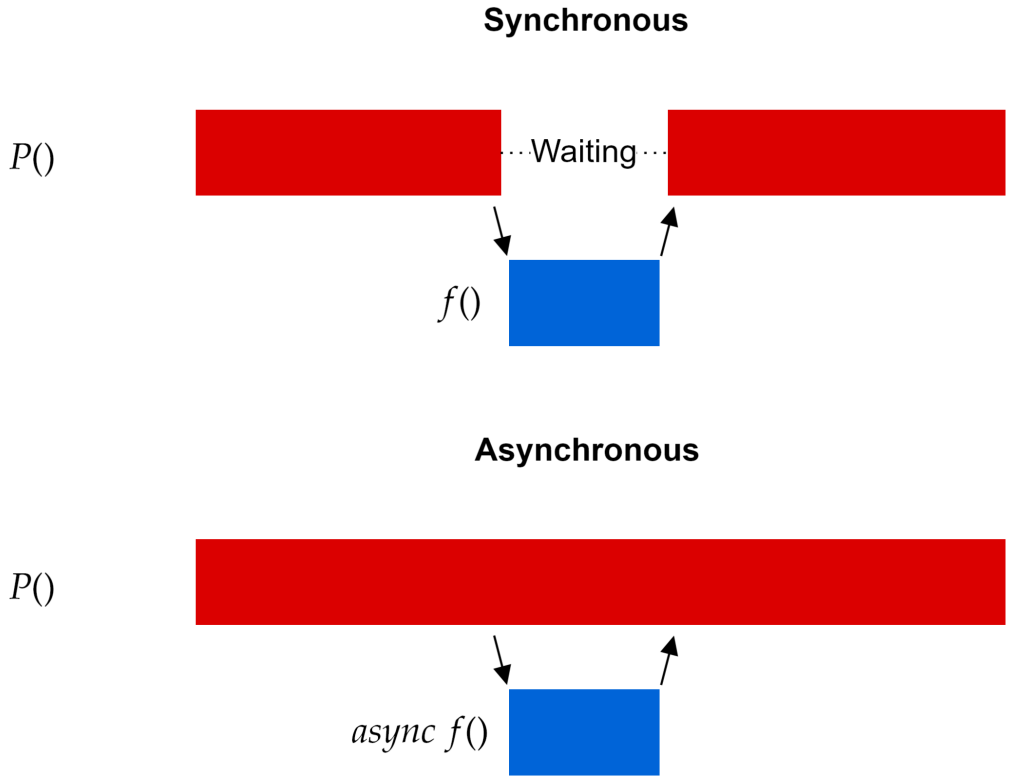


Figure 1.10: Synchronous vs. Asynchronous Function Calls.

- 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:

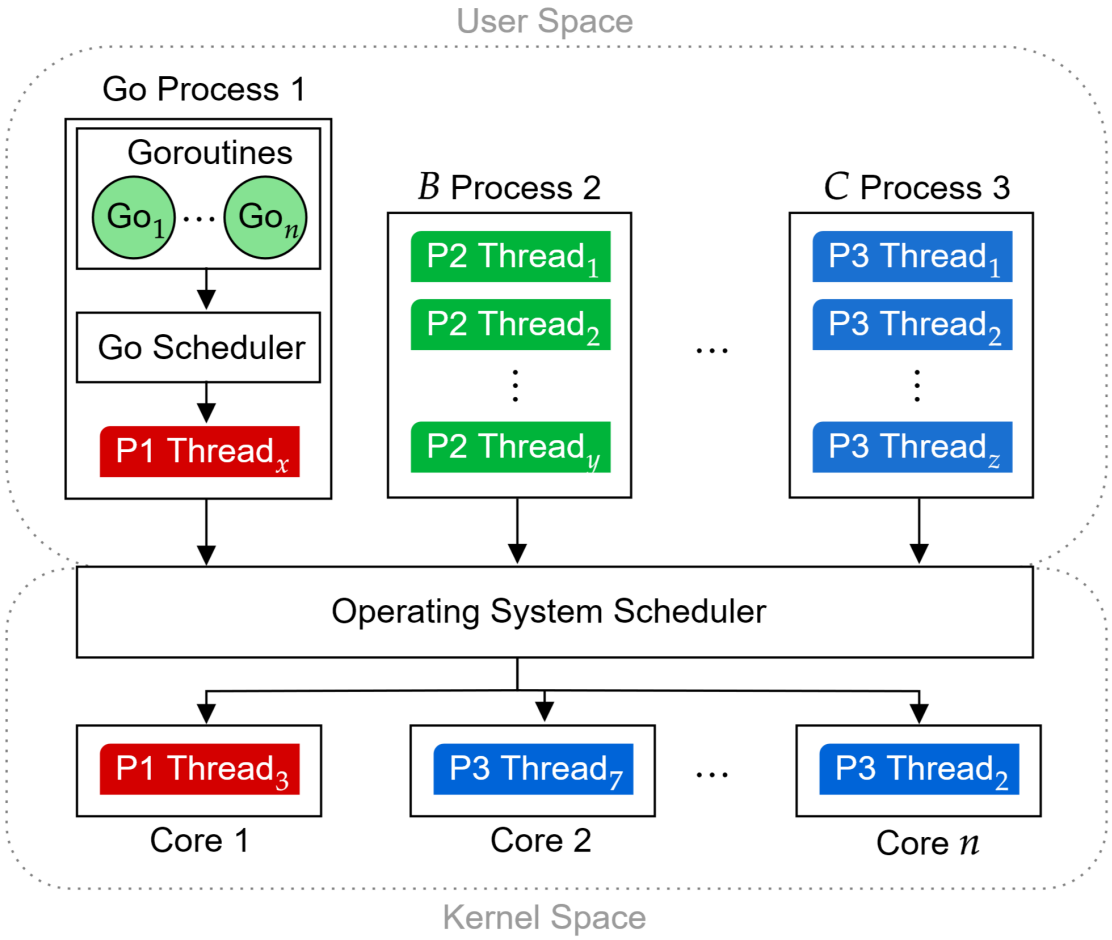


Figure 1.11: Go runtime scheduler with G, M, P entities.

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:

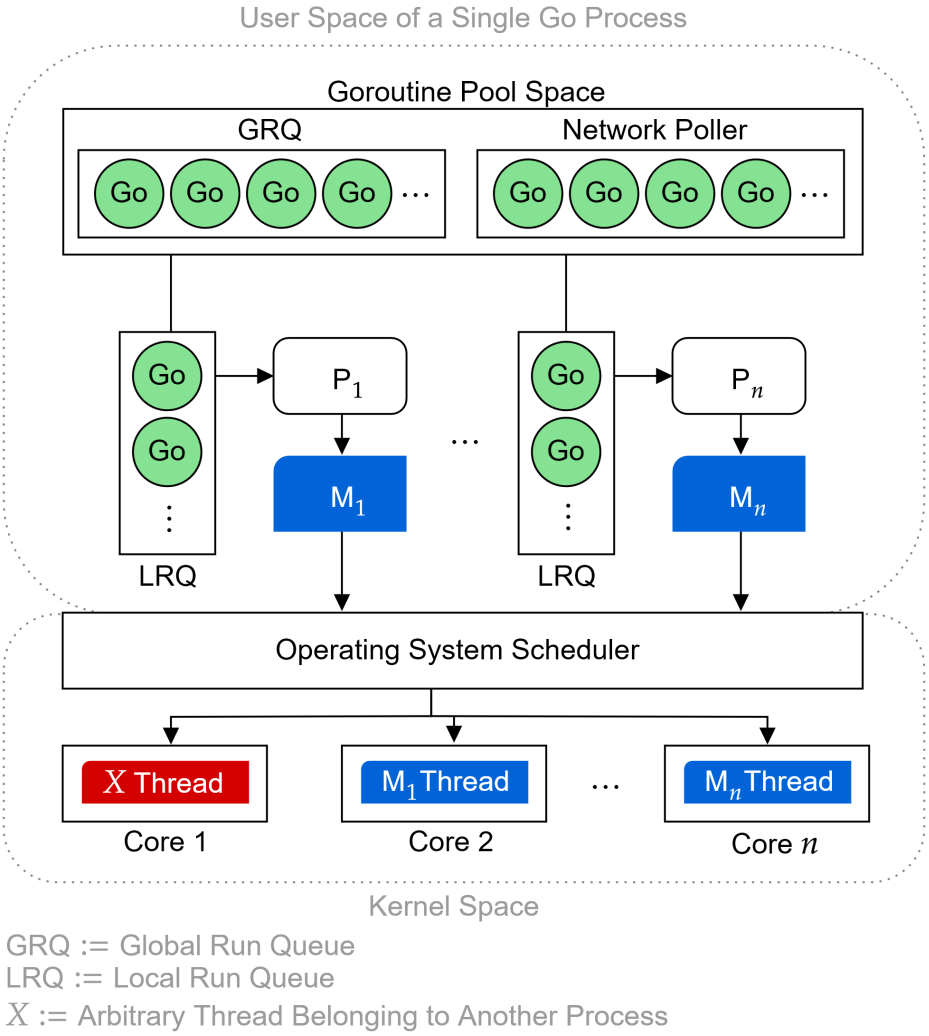


Figure 1.12: Go runtime scheduler within a single process.

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()`.

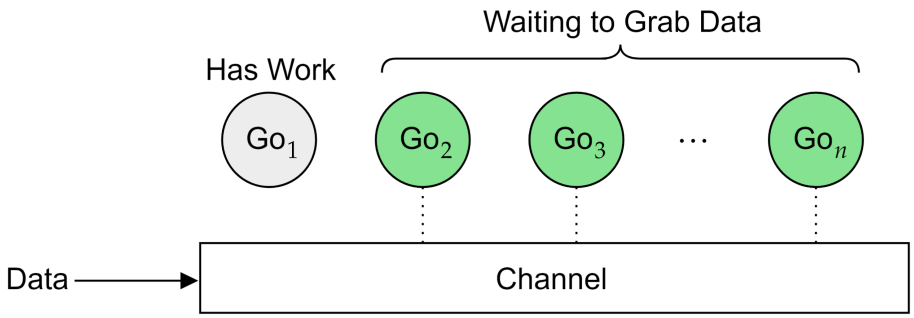


Figure 1.13: A collection of Goroutines competing for the next channel resource.

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. Full channels block the sender, and empty channels block the receiver.

Unbuffered channels undergo a **handshake** process:

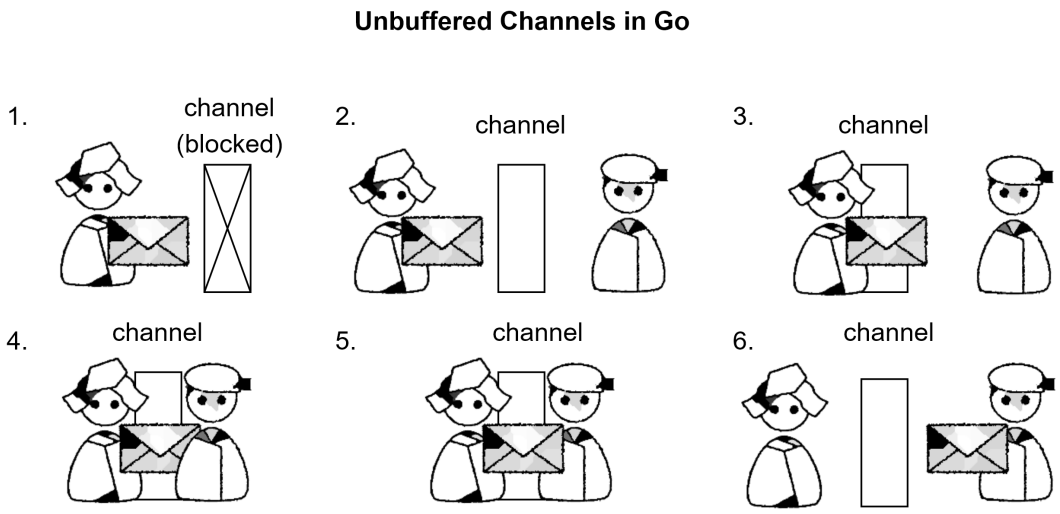


Figure 1.14: Handshake process of an unbuffered channel.

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:

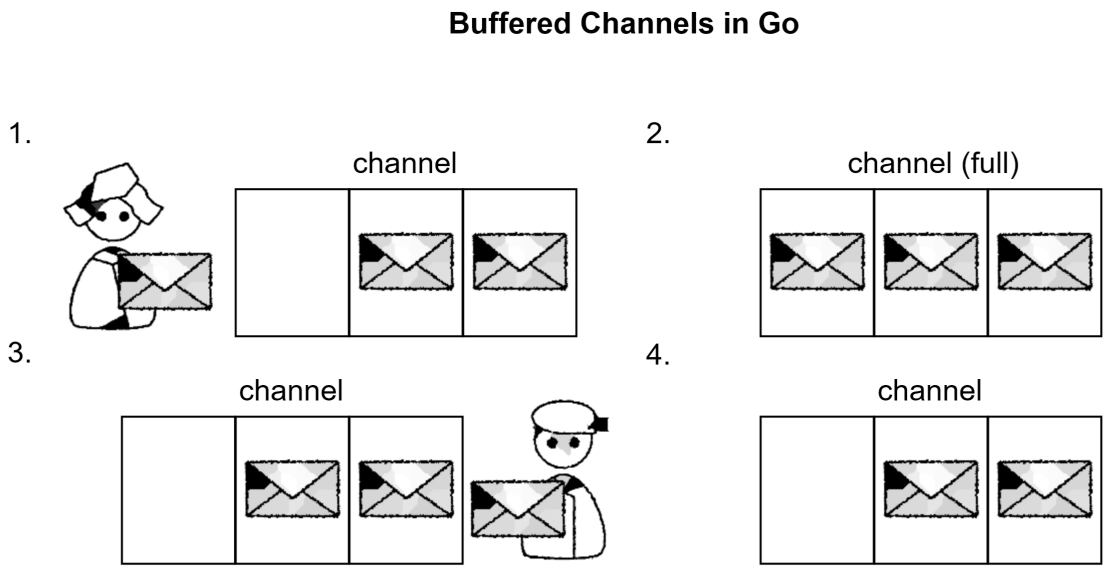


Figure 1.15: Buffered channel allowing for asynchronous communication.

(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:

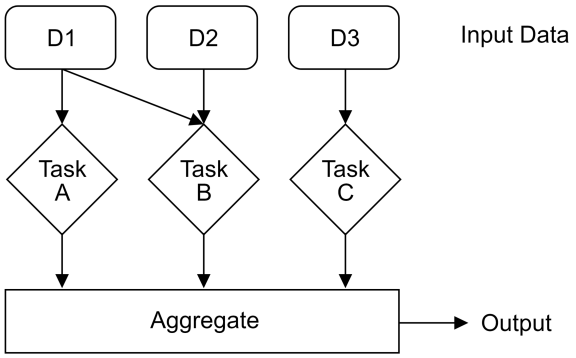


Figure 1.16: Task Parallelism Culminating into an Aggregate Result

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:

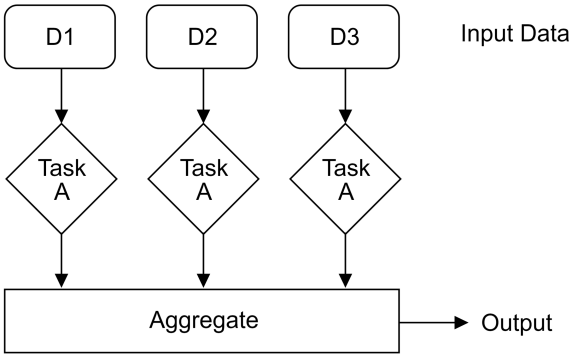


Figure 1.17: Data Parallelism Culminating into an Aggregate Result

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:

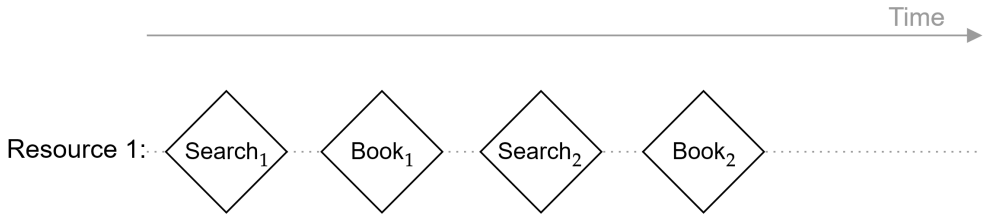


Figure 1.18: Searching and Booking flights concurrently

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

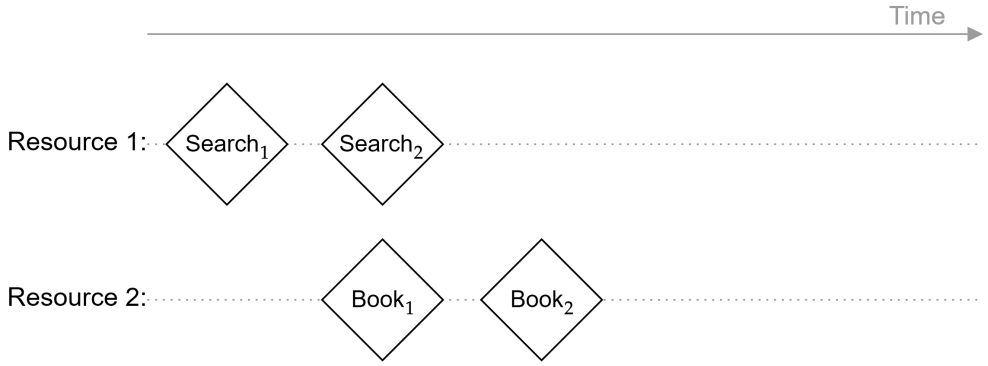


Figure 1.19: Searching and Booking flights in parallel

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:

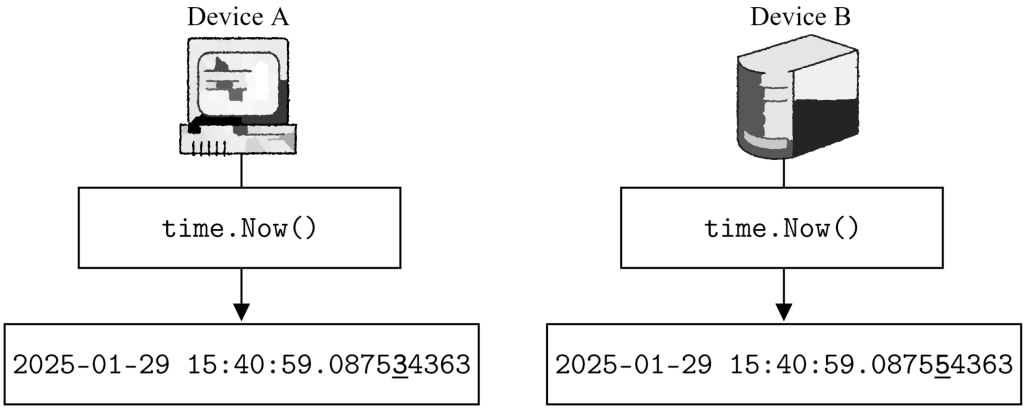


Figure 1.20: Using `time.Now()` on two different machines

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).

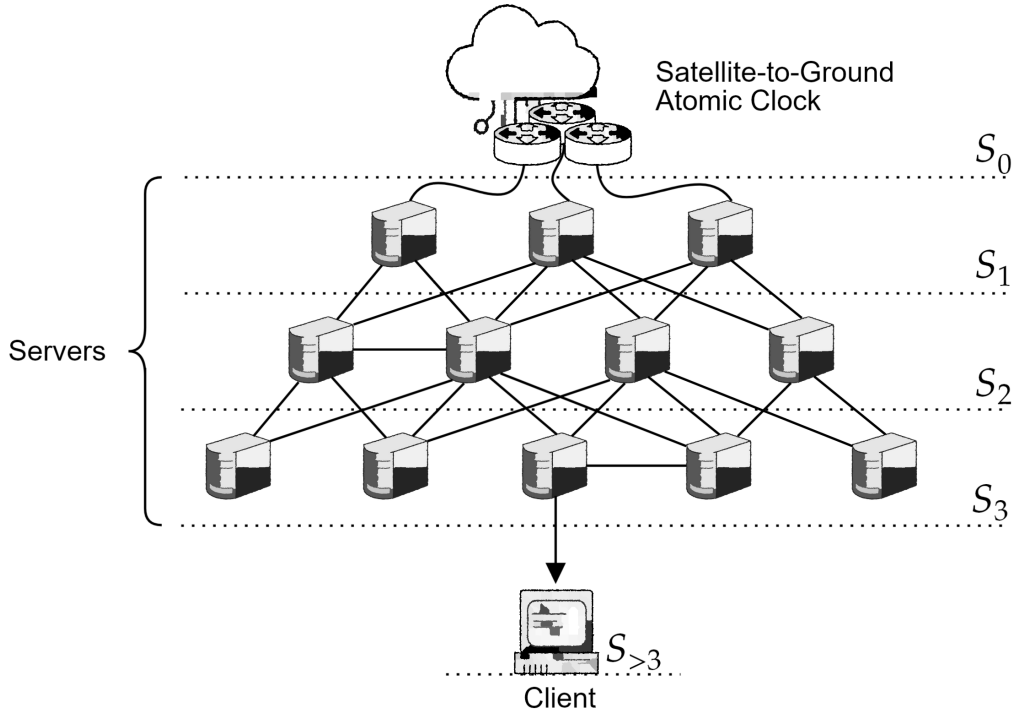


Figure 1.21: NTP Stratum Hierarchy From GPS Satellite Atomic Clock to Client.

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

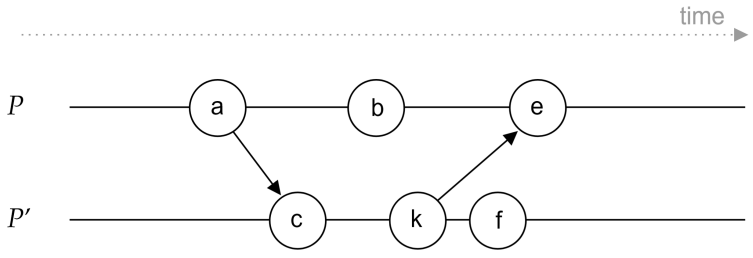
For r **execution trace** (sequence of events), the causal order relationship $\to_r$ is defined as:

- 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

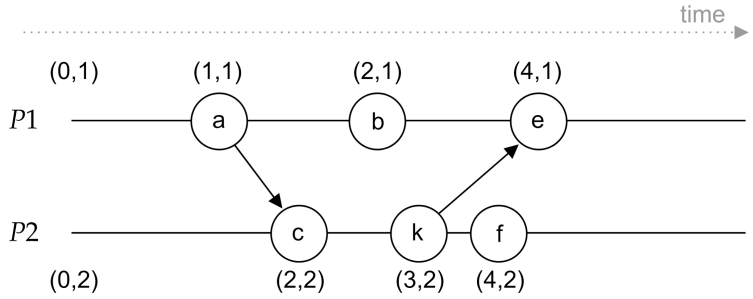
Named after Leslie Lamport, Lamport Clocks assign a logical timestamp to each event:

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 $t(a) < t(b)$.
- Let $a := (3, 1)$ and $b := (1, 2)$, then $t(a) > t(b)$.
- If $t(a) \geq t(b)$, then $a \not\rightarrow_r b$.
- ID ordering break ties: $(1, 1) < (1, 2)$.

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:

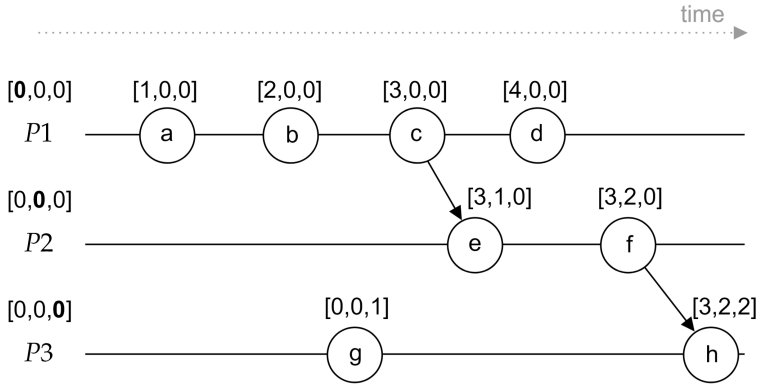
Definition 3.6: Vector Clocks

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 $v(a) < v(b)$.

Theorem 3.6: Total vs. Partial Ordering

Total Ordering means all timestamp pairs are comparable. Hence:

- **Lamport Clocks** define a **Total Ordering** as we can evaluate all pairs of timestamps.
- **Vector Clocks** define a **Partial Ordering** as there are situations such as [1,0] and [0,1], which are non-comparable.

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) $(0, P)$? $(0, Q)$: $P \parallel Q$. To break the tie, the processes would need some order. If P is before Q in process order, then $P \rightarrow Q$ is possible.

(b) $(1, P)$? $(0, Q)$: $P \parallel Q$ or $Q \rightarrow P$ are possible.

(c) $(0, P)$? $(1, Q)$: $Q \parallel P$ or $P \rightarrow Q$ are possible.

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

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

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

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

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

1.4 Implementing RPCs with Go

Before we can make RPC calls in Go, we need to understand Go's typing system.

1.4.1 Typing in Go

Definition 4.1: Typing in Go

Go is a statically typed language, meaning every variable has a fixed type at compile time. The language provides several built-in types, including: **Basic Types:** `int`, `float64`, `string`, `bool`, and **Composite Types:** `array`, `slice`, `map`, `struct`, `interface`.

A **type** in Go can also be user-defined using the `type` keyword. For example, we can define a custom structure:

```
type Person struct {
    Name string
    Age  int
}
```

A `struct` groups related fields together, allowing us to represent objects with multiple attributes. We can create and use instances of this struct:

```
p := Person{Name: "Alice", Age: 25}
fmt.Println(p.Name) // Outputs: Alice
```

Definition 4.2: Typing Functions in Go

Functions are explicitly typed, meaning parameters and return values must be declared:

```
// Takes two ints and returns an int
func Add(x int, y int) int {
    return x + y
}

// We can shorthand parameters of the same type:
func Divide(x, y int) (int, error) { // Returns an int and an error
    if y == 0 {
        return 0, fmt.Errorf("division by zero")
    }
    return x / y, nil
}
```

Here, `fmt.Errorf()` generates an error message.

Definition 4.3: Methods in Go: Pointers vs. Values

In Go, methods can be defined for both pointer receivers (`*T`) and value receivers (`T`). A method with a pointer receiver can modify the original struct, while a method with a value receiver works on a copy. Consider a struct representing a rectangle:

```go
package main; import "fmt"

type rect struct {
    width, height int
}

// Pointer receiver: Can modify the original struct
func (r *rect) area() int {
    return r.width * r.height
}

// Value receiver: Works on a copy of rect
func (r rect) perim() int {
    return 2*r.width + 2*r.height
}

func (r *rect) setWidth(w int) {
    r.width = w
}

func (r rect) setHeight(h int) {
    r.height = h
}

func main() {

    r := rect{width: 10, height: 5}
    // Converts r.area() to (&r).area() automatically
    fmt.Println("area: ", r.area())  // 50
    fmt.Println("perim:", r.perim()) // 30

    rp := &r
    // Converts rp.perim() to (*rp).perim() automatically
    fmt.Println("perim:", rp.perim()) // 30
    fmt.Println("area: ", rp.area())  // 50

    r.setWidth(20)
    r.setHeight(10)
    fmt.Println("width:", r.width) // 20
    fmt.Println("height:", r.height) // 5
}
```

1.4.2 Go's RPC Package

In Go, we can use the `net/rpc` package to implement server-clint RPCs.

Definition 4.4: Setting Up an RPC Server and Client in Go

Setting Up an RPC Server: To create an RPC server in Go, we need to,

- Define a **service type** (a struct) that will handle remote calls.
- Implement **methods** that satisfy the RPC requirements.
- Register the service using `rpc.Register()`.
- Set up a network listener using `net.Listen()`.
- Handle requests using `rpc.HandleHTTP()` and `http.Serve()`.

RPC Method Requirements: Methods must satisfy these constraints or be ignored:

- The method itself and its **type** must be exported.
- The method must return a single value of type `error`, and have exactly **two arguments**, both of which must be **exported or built-in types**.
- The method's **second argument must be a pointer**.

E.g. Signature,

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

Setting Up an RPC Client: To communicate with the RPC server, a client must,

- Establish a connection using `rpc.DialHTTP()`.
- Call remote methods synchronously with `client.Call()`.
- Call remote methods asynchronously with `client.Go()`.

Example 4.1: Simple Local Arithmetic RPC (Part 1)

Let's create a simple Arithmetic RPC. First let's setup our file structure:

```
arith/
├── server/
│   └── server.go
└── client.go
```

■

Example 4.2: Simple Local Arithmetic RPC (Part 2)

Now to setup the server:

```
package main; import "fmt"; import "log"; import "net"; import "net/rpc"

    type ArithService int  // Define the service type

    type Args struct {  A, B int } // Define the arguments struct

    type Quotient struct { Quo, Rem int } // Reply struct for Dividing

    // Both Multiply and Divide methods satisfy the RPC requirements
    func (t *ArithService) Multiply(args *Args, reply *int) error {
        *reply = args.A * args.B
        return nil
    }

    func (t *ArithService) Divide(args *Args, quo *Quotient) error {
        if args.B == 0 { return fmt.Errorf("divide by zero")}
        quo.Quo = args.A / args.B
        quo.Rem = args.A % args.B
        return nil
    }

    func main() {
        arith := new(ArithService)
        err := rpc.Register(arith)
        if err != nil { log.Fatal("Register Error:", err) }

        listener, err := net.Listen("tcp", ":8080")
        if err != nil { log.Fatal("RPC Start Error:", err)}

        defer listener.Close()
        log.Println("Arithmetic RPC server is running on port 8080")

        for {
            // Server waits until a connection is made
            conn, err := listener.Accept()
            if err != nil {
                log.Println("Connection Error:", err)
                continue
            }
            // Serves the connection on a new goroutine
            go rpc.ServeConn(conn)
        }
    }
```

■

Example 4.3: Simple Local Arithmetic RPC (Part 3)

After the following client code, run the server first, then client on a separate terminals:

```go
package main; import "fmt"; import "log"; import "net/rpc"

// Define the same structures as the server
type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

func main() {
    // Connect to the RPC server
    client, err := rpc.Dial("tcp", "localhost:8080")
    if err != nil {
        log.Fatal("Error dialing:", err)
    }
    defer client.Close()

    // Perform a multiplication RPC call
    args := &Args{7, 8}
    var reply int
    err = client.Call("ArithService.Multiply", args, &reply)
    if err != nil {
        log.Fatal("Multiply error:", err)
    }

    // Perform a division RPC call
    var quotient Quotient
    err = client.Call("ArithService.Divide", args, &quotient)
    if err != nil {
        log.Fatal("Divide error:", err)
    }

    // Output: 7 * 8 = 56
    fmt.Printf("ArithService: %d * %d = %d\n", args.A, args.B, reply)

    // Output: 7 / 8 = 0, remainder = 7
    fmt.Printf("ArithService: %d / %d = %d, remainder = %d\n", args.A,
        args.B, quotient.Quo, quotient.Rem)
}
```

■

2.1 Saving System State: Snapshots

This section discusses saving state. This is useful for fault-tolerance and system migrations.

Definition 1.1: Snapshot

A **snapshot** is a consistent global state of a distributed system at a specific point in time.

Definition 1.2: Consistent vs. Inconsistent Snapshots

To evaluate a snapshot's consistency, we compare events in the system **pre-snapshot** (events before the snapshot) and **post-snapshot** (events after the snapshot). The snapshot itself is instantaneous, like a photograph. Given an event ordering r:

- **Consistent Snapshots**: Respect causal dependencies. Let there be events e_1 and e_2; If $e_1 \rightarrow_r e_2$, then e_1 must be included in the snapshot if e_2 is present.
- **Inconsistent Snapshots**: Violate causal dependencies. If e_2 is included without the causally preceding event e_1, then the snapshot is inconsistent.

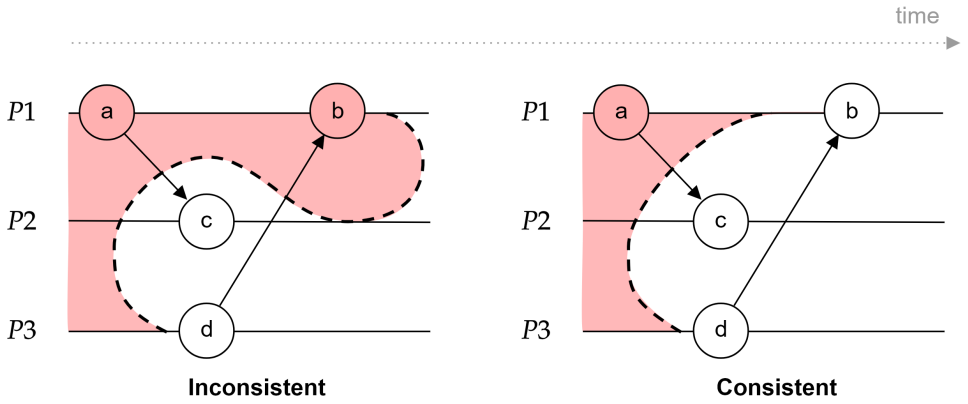


Figure 2.1: Inconsistent vs. Consistent Snapshots (pre-snapshot highlighted in red)

Here, in the inconsistent snapshot, b is included without d, its causally preceding event. In the consistent snapshot, only a is included. In this snapshot c and d could be added without violating causality.

There are many snapshot protocols, but we will focus on the **Chandy-Lamport Algorithm**.

Definition 1.3: Chandy-Lamport Algorithm

The **Chandy-Lamport Algorithm** is a snapshot algorithm that is used in distributed systems for recording a consistent global state of an asynchronous system. In this protocol:

- The snapshot procedure does not disrupt other processes.
- Each process records its local state.
- Any process can initiate the snapshot.

This model requires:

- **No Failures**: No failures during the snapshot.
- **First in, First out Channels (FIFO)**: no lost or duplicated messages.
- **Strongly Connected Network**: All processes can reach every other process.
- **Single Initiator**: Only one process can initiate the snapshot.

The initiator then:

- Sends a **Marker** message to all outgoing channels.
- Records local and incoming channel data.

Recipients of the marker message:

- Designates the channel which the marker arrived as **empty**.
- Records their local state and all other incoming channels except the empty one.
- Sends the marker to all outgoing channels.

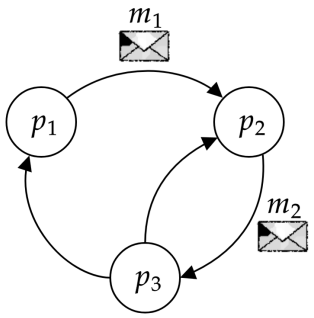
Completion: When all processes have received and sent a marker, the snapshot is complete. This means every processes incoming channel is empty, hence the conclusion of the snapshot.

Important Notes:

- Subsequent markers received are ignored, **only the first** marker received is acted upon.
- Without the FIFO property, restarting the system may not be possible, as it relies on the order of messages to reproduce state.

Observe the following illustration of the Chandy-Lamport Algorithm given processes p_1, p_2, p_3:

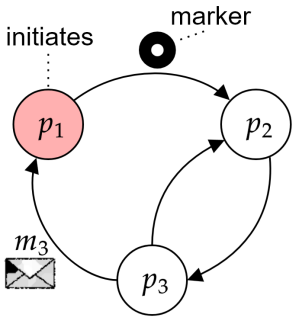
(1)



Snapshot

p_1	p_2	p_3

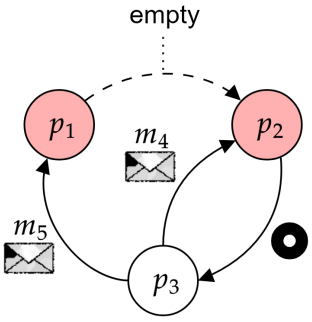
(2)



Snapshot

p_1	p_2	p_3

(3)



Snapshot

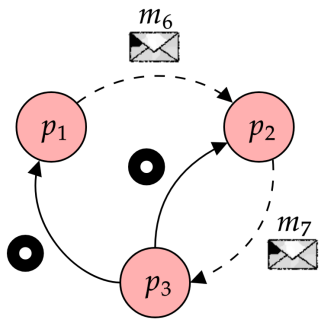
p_1	p_2	p_3
m_3		

Examining these first three steps: (1) The system before the snapshot. (2) p_1 initiates the snapshot sending a marker to p_2 and recording its local and incoming channel's state. (3) p_2 receives the marker, designating the incoming channel from p_1 as empty, sends the marker on outgoing channels, and begins recording. We also see, p_1 has recorded an incoming message m_3 from (2).

We continue on the next page.

We continue the snapshot process with the following steps:

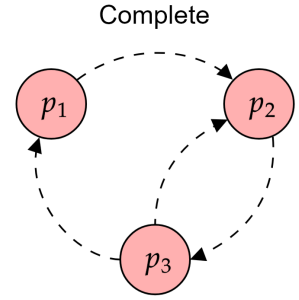
(4)



Snapshot

p_1	p_2	p_3
m_3 m_5	m_4	

(5)



Snapshot

p_1	p_2	p_3
m_3 m_5	m_4	

(4) p_3 received the marker from p_2, and designates that incoming channel as empty. It begins recording state, sending out the marker to p_1 and p_2. Take notice that messages m_4 and m_5 have been recorded from (3). (5) All channels are now empty, concluding the snapshot. Take note that m_6 and m_7 were not recorded as they were sent on empty channels.

Theorem 1.1: Replay of a Snapshot

When replaying a snapshot captured using the Chandy-Lamport protocol, messages in transit (i.e., sent by a task whose state was recorded, but not yet received by others) **must also be replayed**.

In particular, if task A sends a message to task B, and only A is included in the snapshot, then the message is considered in transit and must be delivered to B upon replay.

Consider the following illustrations and think about how many messages may be replayed:

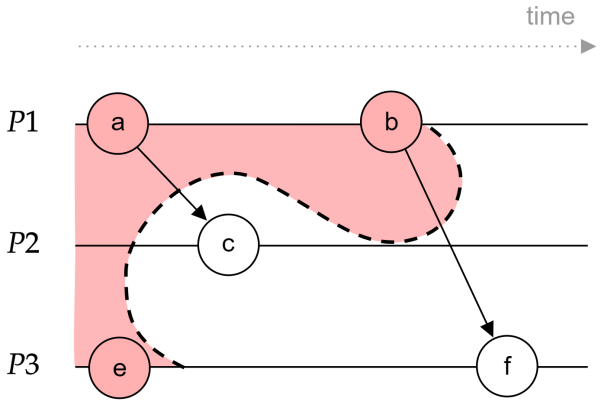


Figure 2.2: tasks a, b and e are recorded in the snapshot. Both a and b send messages from which aren't recorded in the snapshot. After reloading the snapshot, a and b's messages **must be replayed**.

2.2 Replication: Synchronizing State

This section discusses replicating state across distributed systems.

Definition 2.1: Replication

Replication is the process of maintaining multiple copies of the same data on different nodes (machines). This is done for fault-tolerance, load balancing, and data locality.

Problem Space:
Consider we are running a money transfer service, from which Alice and Bob interact with:

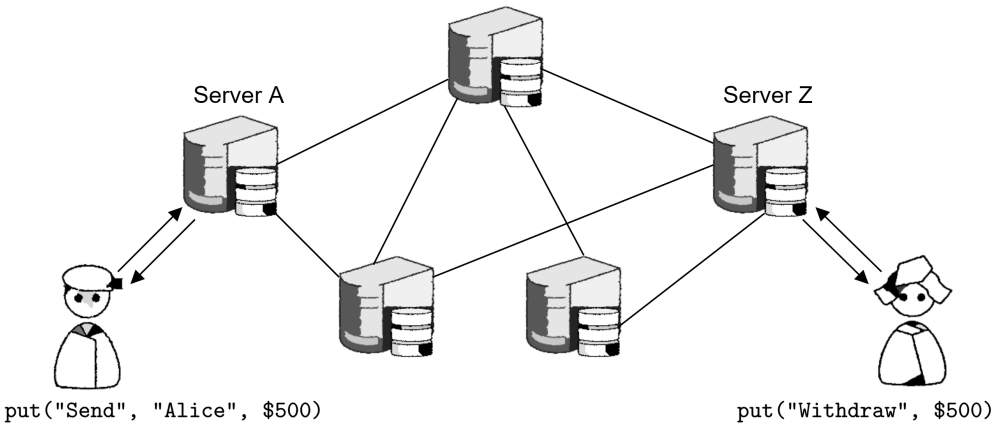


Figure 2.3: Bob sending money to Alice, while she withdraws money on two different servers.

In this scenario Many problems can arise: What if,

- The respective servers crash while or after Alice or Bob make requests?
- Alice and Bob share the same account, how do we ensure consistency?

In all, how do we ensure the propagation and synchronization of state across multiple servers? We consider two models:

Definition 2.2: Active vs. Passive Replication

Active Replication: Client sends requests to all servers at the same time and waits for acknowledgments. This method must ensure that all requests are processed in the same order (expensive).

Passive Replication: Client sends requests to a primary server, which then forwards the request to backup servers.

We'll be move forward with **Passive Replication** as the preferred choice in this section. Though it is less expensive than active replication, it still has its challenges:

- **Consistency**: How do we ensure all backups are consistent?
- **Failure Handling**: What if the primary server fails?
- **Performance**: How do we ensure that the system is performant as we scale backups?

We consider two methods of replication:

Definition 2.3: State vs. Request Replication

State Replication: Forward the entire state to backups. This results in large message sizes, but is relatively simple depending on the system.

Request Replication: Forward only requests to backups. This results in smaller message sizes, but adds complexity when requests are not deterministic (e.g., random number generation).

Still again, we run into the following problem:

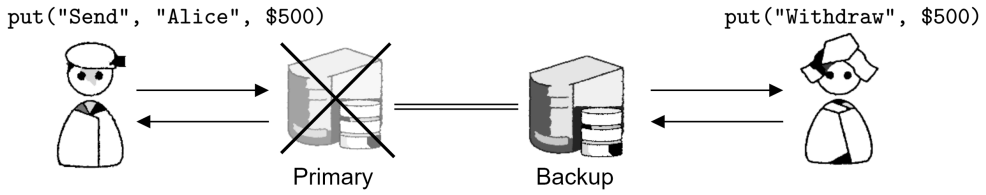


Figure 2.4: Bob's primary server failing, as Alice accesses the backup server.

How do we ensure both parties receive consistent feedback, even when the primary server fails?

Definition 2.4: Commit Point

The point at which the client is committed to a transaction, goes as follows:

1. The client sends a request to the primary server and waits for an acknowledgment.
2. The primary server forwards the request to the backup servers.
3. The backup servers process the request and send an acknowledgment to the primary server.
4. The primary server sends an acknowledgment to the client.

Step 4 is considered the **commit point**.

The following Figure illustrates the commit point in action:

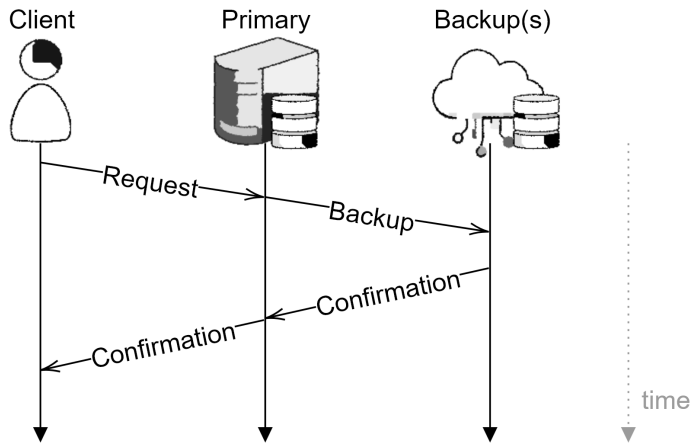


Figure 2.5: Client's requests propagating through the primary and backup sever.

We now can confirm that the client's request has been processed by the primary and backup servers, though this creates a lot of overhead as we scale the number of backups.

Now that some level of consistency is ensured, we want to orchestrate designating a new primary server when the current primary server fails:

Definition 2.5: Arbitration Server and Primary Election via Test-And-Set

In a distributed system with a primary-backup model, maintaining a **single active primary** is crucial. When a primary server fails, a **backup must be promoted**, but only if the failure is confirmed. An **Arbitration Server**, also called **Configuration Service Provider (CFG)** ensures a controlled **failover** (switching to a backup) process using **test-and-set**. It goes as follows:

- If the **primary fails** (or becomes unreachable), the CFG.
- The backup executes a `test_and_set(f,1)` operation on a shared variable f.
  - If $f = 0$, the backup sets $f = 1$ and becomes the new **primary**.
  - If $f = 1$, another server has already taken over, preventing duplicate primaries.
- Clients redirect their requests to the new primary.

To demonstrate test-and-set in action, consider the following illustration:

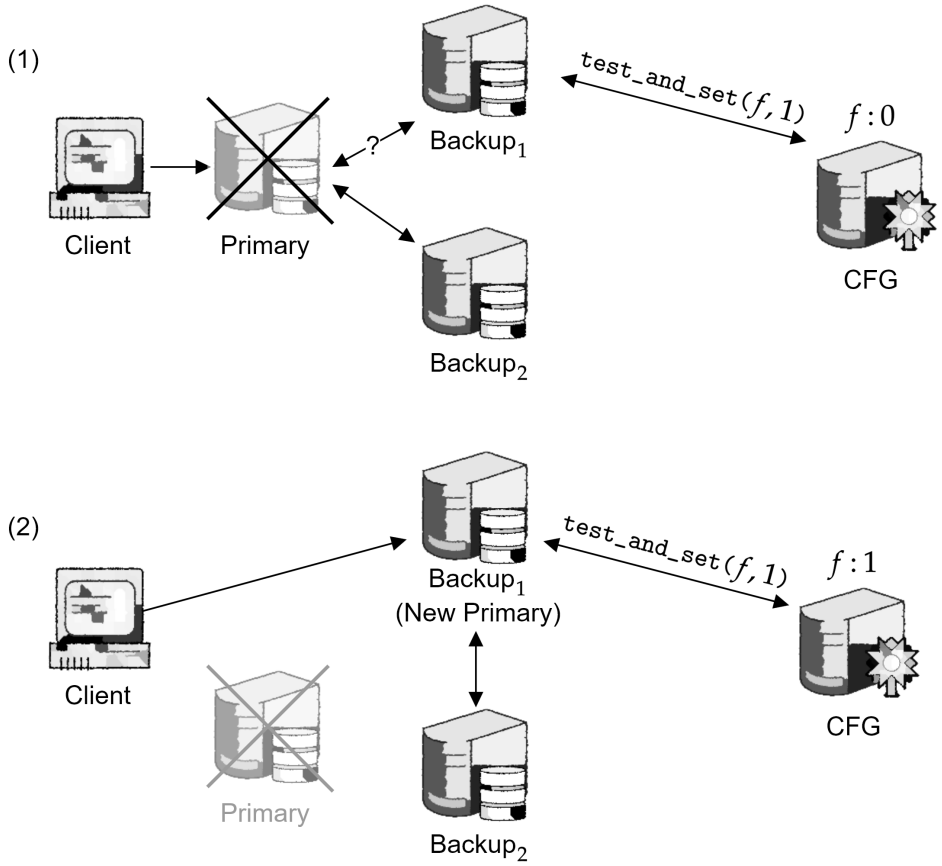


Figure 2.6: Backup server 1, is unable to connect to the primary, sends promotion request to CFG. All servers are connected to the CFG with access to the shared variable f. While the primary server is reachable, $f = 0$. When the primary fails, the backup server executes `test_and_set(f,1)`.

A mutex is crucial to avoid data-races with multiple servers attempting promotion:

```
var flag int = 0; var mu sync.Mutex;
func (s *Server) TestAndSet(f *int, reply *int) error {
    mu.Lock()
    *reply = 0
    if flag != *f {
        flag = *f
        *reply = 1
    }
    lock.Unlock()
    return nil
}
```

Now what if the primary server partially processed a requests before failing? Consider the following:

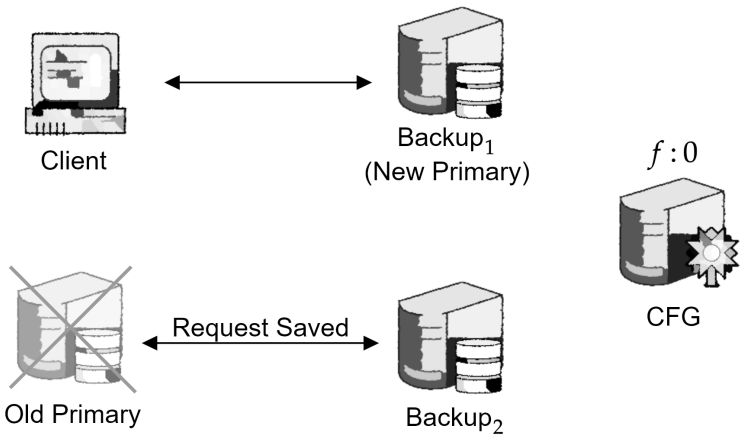


Figure 2.7: Primary server failing after partially processing a request.

To combat this issue, we discuss the **Chain Replication** method:

Definition 2.6: Chain Replication

The **Chain Replication** architecture servers are organized in a linear sequence, forming a chain: $s_1, s_2, \ldots, s_n$, where s_1 is the **head** and s_n is the **tail**:

- **Write Operations**: Clients send write requests to the head (s_1), which is forwarded to s_2, and so on, until the tail (s_n).
- **Read Operations**: Clients directly read from the tail (s_n), ensuring fully processed states; Specifically as s_i states may not be fully propagated through the chain.

Failover Handling:

- If server s_i fails, s_{i-1} forwards to the successor s_{i+1} instead.
- If the head (s_1) fails, the CFG promotes s_2 to be the new head.
- If the tail (s_n) fails, the CFG designates s_{n-1} as the new tail.

Tip: OSDI 2004 featured Robbert van Renesse and Fred B. Schneider from Cornell University, presenting Chain Replication. Van Renesse, originally from the Netherlands, and Schneider, from the U.S., have significantly influenced distributed systems research.

Consider the comparison between **Primary-Backup** vs. **Chain Replication**:

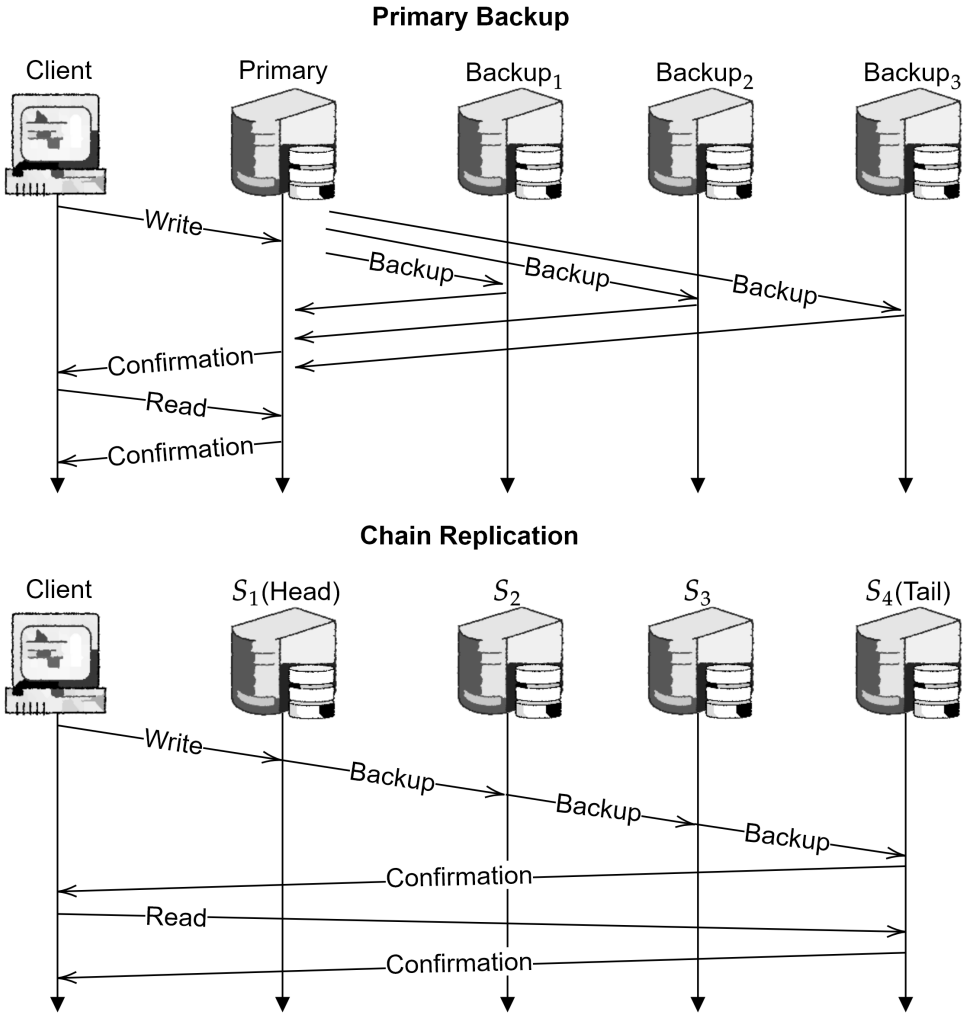


Figure 2.8: Primary-Backup vs. Chain Replication

Theorem 2.1: Primary-Backup vs. Chain Replication

In terms of **Latency**, **Primary-Backup** is faster as the request can theoretically be processed in parallel. However, **Chain Replication** is more **Fault-Tolerant**. If Reads/Writes are 50/50, then Chain Replication has **faster throughput** as it can split up reads (to the tail) and writes (to the head).

The Big Problem with Chain Replication:

There are some major assumptions that the Chain Replication Protocol makes:

- **Fail-Stop Servers**: Failures are complete and detectable (does not address slow servers).
- **Centralized Configuration Service**: Assumed never to fail.
- **Failure Detection**: Failures are easy to identify; **However**, distinguishing temporary vs. permanent failures is challenging.
  - Quick failure detection prevents stalls.
  - Over-eager detection may cause unnecessary data copying.

We will address these in the next section, discussing the **Raft Consensus Algorithm**.

2.3 Raft Protocol: Consensus Replication

As we've discussed so far, replication consistency is crucial and a difficult task. The next strategy, **Raft**, deals with such problem. There are other solutions, however, Raft is considered safe and easier to implement correctly than other solutions:

Definition 3.1: State Machines & Consensus Algorithms

A **State Machine** processes sequence of inputs from a **log** and saves them in state. **Replicated state machines** are implemented via **replicated logs** across multiple servers utilizing a **Consensus Model**, which facilitate data agreement between nodes.

Hence, commands need be **deterministic**, and the consensus algorithm **must be** flexible enough to handle and replicate application specific data.

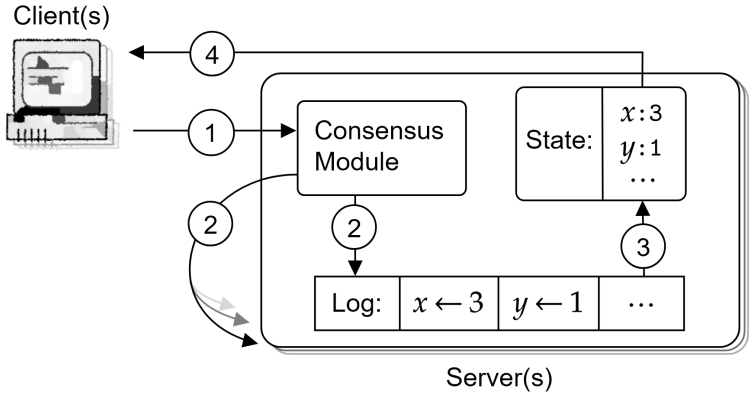


Figure 2.9: High-level framework for replicated state machines.

In the above the **Consensus Module** communicates with other servers to serve consistent logs.

2.3.1 Procedure Outline: Heartbeats, Elections, & Log Replication

Now to define the parts which make up a consensus algorithm:

Definition 3.2: Consensus Algorithm Components

A **Consensus Algorithm** involves the following components:

- **Safety**: Always returns correct results in spite of, network delays, partitions, duplications, and reorderings.
- **Liveness**: A **Cluster** (group of servers) must tolerate a subset of server failures (e.g., A cluster of 5 servers can tolerate 2 failures). Offline servers may later recover and rejoin the cluster.
- **Time Agnostic**: The algorithm must not rely on synchronized clocks.
- **Majority Rule**: A majority of servers must agree on a value before it is committed. Minority slow servers must not block the system.

At a high level, The Raft Algorithm:

Definition 3.3: Raft Abstract

The Raft Algorithm involves three main components:

- **Leader Election**: A leader ℓ is elected to manage the replication process of backups β.
- **Heartbeats**: Where ℓ and β exchange consistent pulses of data to ensure liveliness.
- **Assurance**: Commit points (2.4) are established between the client, ℓ, and β.

In Raft, servers are given roles to manage the replication process.

Definition 3.4: Raft Server States

A Raft server can be in one of the following states:

- **Follower**: A server that listens to the leader.
- **Candidate**: A server that is running for leader.
- **Leader**: A server that is managing the replication process.

Followers are passive and simply listen to the leader.

Definition 3.5: Raft Heartbeats

Raft servers exchange **heartbeats** to ensure liveliness. Each server on boot is randomly assigned a timeout value. After the initial timeout, a server sends a signal akin to the out-and-in beats of a heart. **Out-beats** are origin signals, and **in-beats** are return signals. If a server β receives an out-beat from another server ℓ before its in-beat, the β timer resets and follows ℓ's beat. The leader of the beat's timer is not used.

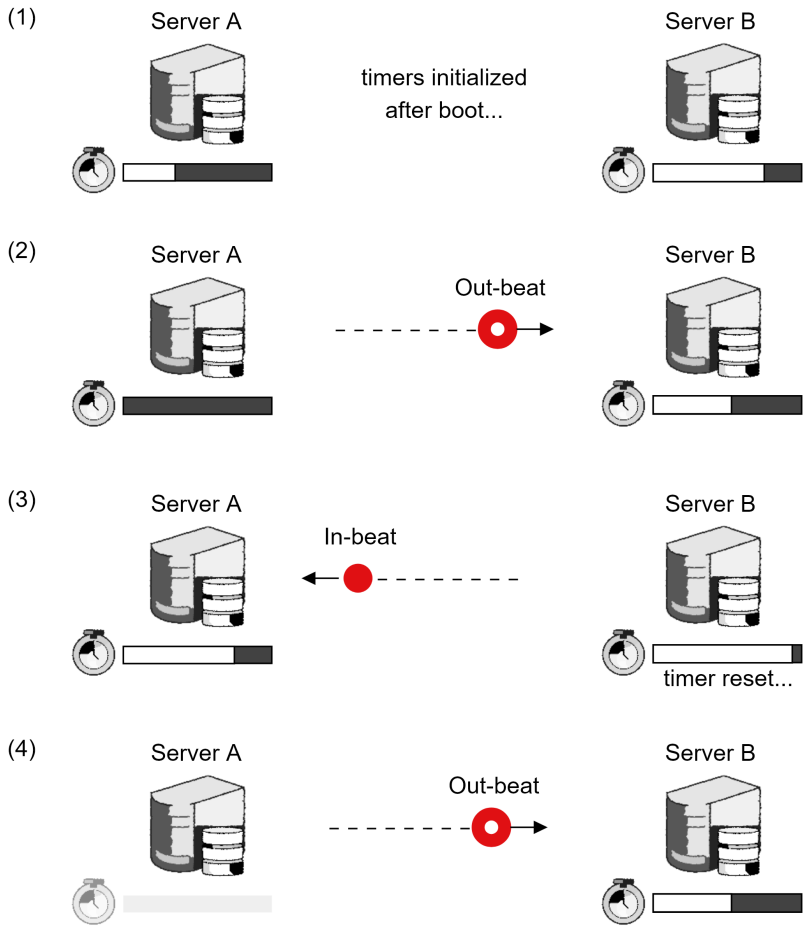


Figure 2.10: Server A and B exchanging heartbeats after system initialization (1). Server A's timer runs out first, sending the first signal to B (2). B has received A's out-beat, and now follows them. B resets their timer, and returns the signal to A (3). A's timer isn't effect, **nor is it used** whilst a leader of the beat (4).

Definition 3.6: Raft Election Terms

Let us denote an arbitrary server as β, then β is either of class γ (**Follower**), ς (**Candidate**), or ℓ (**Leader**). I.e., we have types $\{\beta : \gamma \mid \varsigma \mid \ell \}$ The Raft Election Process involves the following:

1. **Timeout ($\gamma \to \varsigma$):** First, all β initialize as γ with a random timer (e.g., 150-300ms). Once the timer runs out, γ transitions to ς.
2. **Candidate Election ($\varsigma \to \ell$):** ς votes for itself and sends a **RequestVote RPC** to all β, informing them of the new term and approval request. All β vote once per term for the RPC they received first. If ς receives the majority vote, it transitions to ℓ.
3. **Split Vote:** If all top ς tie in votes, all ς timeout, waiting for the next term. At this point, if a γ's timer runs out before the ς turn-around, then $\gamma \to \varsigma$ starting a new term election.
4. **Behind Leaders & Candidates ($\ell \to \gamma, \varsigma \to \gamma$):** If a ℓ or ς ever receives a heartbeat from a higher term β, it reverts to γ. Additionally, lower term signals are **ignored**.
5. **Leader Timeout ($\gamma \to \varsigma$):** If γ does not receive a heartbeat from the ℓ (server failure) whilst their timer runs out, they transition to ς and start a new election.

Let's observe a cluster of three servers:

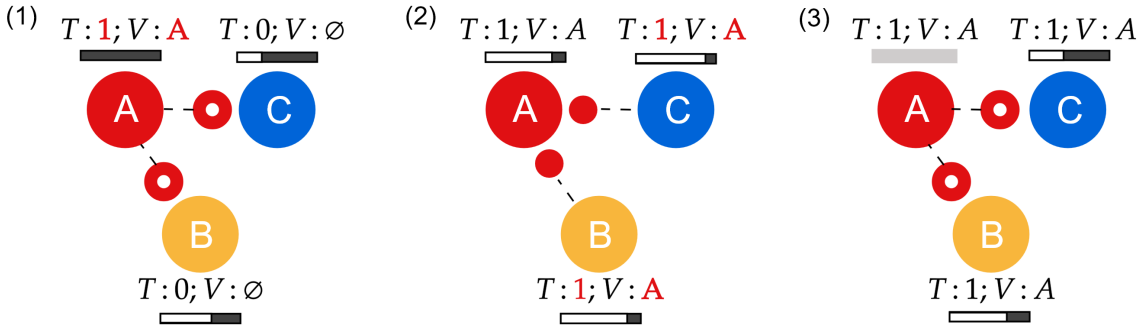


Figure 2.11: First Election after boot where A, B, and C nodes in a cluster of three have been initialized. (1) A times out first, becoming a Candidate and sending a RequestVote RPC to B and C. (2) B and C vote for A. (3) A receives the majority vote and becomes the Leader. Now A's timer is no longer used.

Observe a cluster of four who is dealing with a split vote:

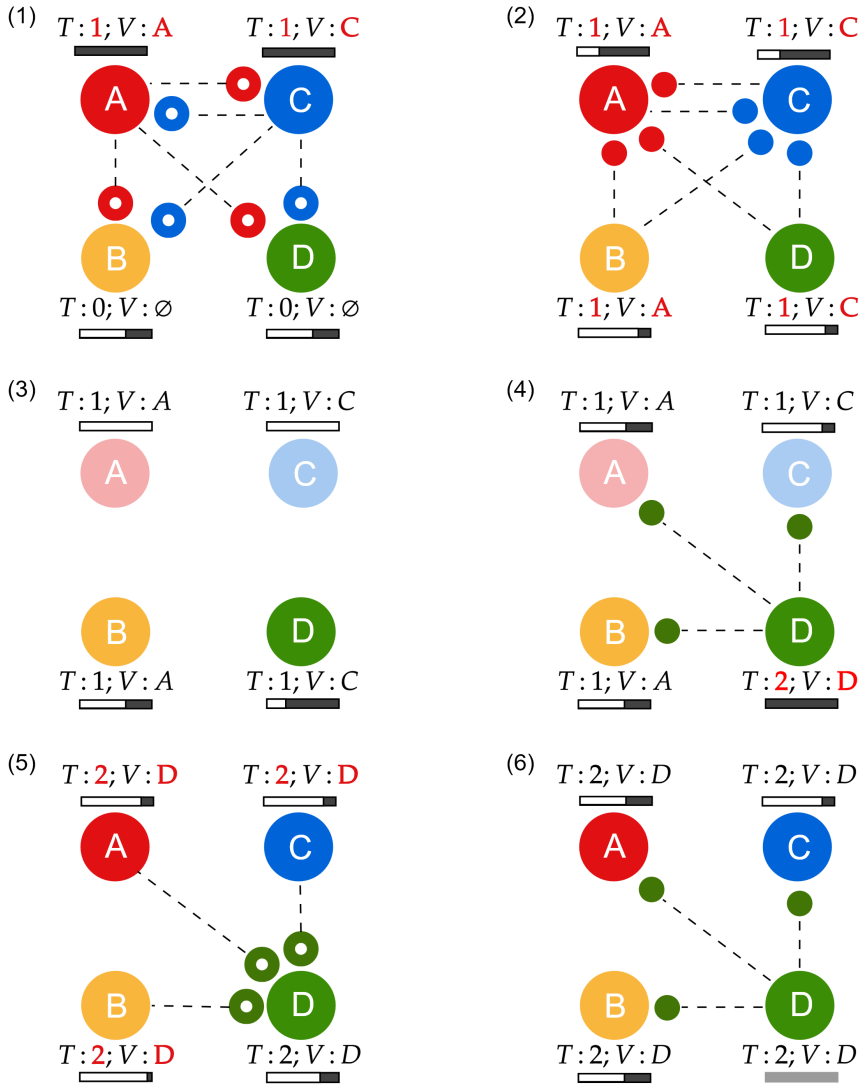


Figure 2.12: (1) A and C's timers run out first, sending out a RequestVote RPC to all other servers. (2) B votes for A, and D votes for C. (3) A and C timeout as per split vote. (4) D times out starting a new term, casting their Request RPC. (5) All servers respond to the higher order term. (6) D becomes the Leader (Their time no longer used).

Definition 3.7: Log Replication

Given { β (Server) : γ (Follower)| ς (Candidate)| ℓ (Leader) }, the Raft Log Replication Process involves the following:

- **Leader Log Replication:** Once a ℓ is elected, it services command requests from the client. The ℓ first appends the command to its **indexed log**, then sends an **AppendEntries RPC** in parallel to all β. Once a majority of β have replicated the log, the command is **committed**, and thus, safe to apply to their state machine.
- **Order of Execution:** The ℓ maintains a `nextIndex[]`, indicating the next expected log entry for all β. The ℓ sends missing log entries to lagging β; otherwise, it sends empty **AppendEntries RPCs** as heartbeats.
- **Leader Redirection:** All β that receive client requests, redirect the client to the ℓ.

Theorem 3.1: Log Matching Property

If two entries in different logs have the same index and term, then:

- They store the same command.
- All proceeding entries are the same.

Definition 3.8: Log Correction

Given { β (Server) : γ (Follower)| ς (Candidate)| ℓ (Leader) },

If ℓ fails, the new ℓ may be missing log entries. Let i denote the last index of ℓ's log entry, and j the last index of other βs' logs. If after an **AppendEntries RPC**, β's $j \neq i$, then ℓ decrements β's **nextIndex** to $(j-1)$ **per call**, until $j = i$. Once majority coherence is met, the log is committed.

This holds as we assume the preceding Theorem (3.1) is true. This is to say, the leader has a—**Append Only Property**—that it does not modify its own entries, but **only** appends new ones.

Note: A slight optimization for log correction, is for γ to tell ℓ the conflicting term and its first index. With that, ℓ can skip the log entries of such term. Though this isn't necessary as in real world applications, these conflicts are infrequent.

2.3.2 Safety: Restricting Leader Election

Since previously discussed, a new leader may be missing log entries, from which they tell other servers to discard mismatches. This can become problematic when the previous leader and other servers have committed several entries.

To fix this, we introduce constraints on leader election:

Definition 3.9: Leader Completeness

All proceeding leaders must have all committed entries of the previous leader.

Definition 3.10: Leader Election Restriction

Given { β (Server) : γ (Follower)| ς (Candidate)| ℓ (Leader) },
New ℓ must demonstrate **Leader Completeness**. To ensure this, β upon receiving a **RequestVote RPC** from ς checks:

- If ς's last log is from a lower term, the vote is **rejected**.
- If ς's last log is from the same term, but shorter, the vote is **rejected**.
- If ς's last log is from a higher term, then the vote is **accepted**.
- If ς's log is **at least** as up-to-date as the voter, the vote is **accepted**.

Theorem 3.2: Present Term Commitment

Leaders cannot assume previous term entries are committed, as they may not have been fully replicated. Hence they only commit entries from their term, and by the **Log Matching Property** (3.1), previous entries are also committed.

Theorem 3.3: Ensuring a Leader Through Timing

To ensure that there is always a leader in the face of probabilistic timing, these magnitude requirements should be satisfied:

$$broadcastTime \ll electionTimeout \ll MTBF$$

$broadcastTime$ measures the average time of heartbeat reciprocation (e.g., .5ms-20ms). Then $electionTimeout$, the randomly assigned timeout times for each server (e.g, 10ms-500ms), and $MTBF$, the average time between failures of a server (e.g., several months or more).

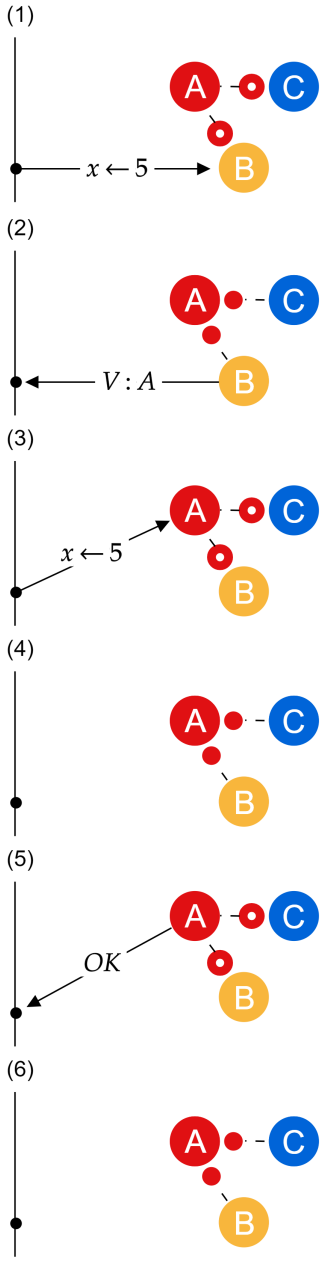
2.3.3 AppendEntries, State, and RequestVote RPC Schema

The next two pages are a summary of the Raft RPCs **AppendEntries**, **RequestVote**, and **state** for implementation:

AppendEntries RPC	
Invoked by leader to replicate log entries; also used as heartbeat.	
Arguments	**Description**
term	Leader's term.
leaderId	Allows follower to redirect clients.
prevLogIndex	Index of log entry immediately preceding new ones.
prevLogTerm	Term of **prevLogIndex**, entry.
entries[]	Log entries to store (empty for heartbeat; may send more than one for efficiency).
leaderCommit	Leader's **commitIndex**.
Results	
term	Current term, for leader to update itself.
success	**true** if follower contained entry matching **prevLogIndex** and **prevLogTerm**.
Receiver Implementation	

1. Reply **false** if **term** $<$ **currentTerm**.
2. Reply **false** if log doesn't contain an entry at **prevLogIndex** whose term matches **prevLogTerm**.
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it.
4. Append any new entries not already in the log.
5. If **leaderCommit** $>$ **commitIndex**, set **commitIndex** $=$ min(**leaderCommit**, index of last new entry).

STATE	
Persistent state on all servers (stored on stable storage before responding to RPCs)	
currentTerm	Latest term server has seen (initialized to 0 on first boot, increases monotonically).
votedFor	Candidate ID that received vote in current term (or null if none).
log[]	Log entries; each entry contains a command for the state machine and the term when it was received by the leader (first index is 1).
Volatile state on all servers	
commitIndex	Index of the highest log entry known to be committed (initialized to 0, increases monotonically).
lastApplied	Index of highest log entry applied to the state machine (initialized to 0, increases monotonically).
Volatile state on leaders (Reinitialized after election)	
nextIndex[]	For each server, index of the next log entry to send to that server (initialized to leader last log index + 1).
matchIndex[]	For each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically).
RequestVote RPC	
Invoked by candidates to gather votes.	
term	Candidate's term.
candidateId	Candidate requesting vote.
lastLogIndex	Index of candidate's last log entry.
lastLogTerm	Term of candidate's last log entry.
Results	
term	Current term, for candidate to update itself.
voteGranted	**true** means candidate received vote.
Receiver Implementation	
1. Reply **false** if **term** < **currentTerm**. 2. If **votedFor** is **null** or **candidateId**, and candidate's log is at least as up-to-date as receiver's log, grant vote.	



	State	Logs		
A	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$		
B	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$		
C	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$		

	State	Logs		
A	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$		
B	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$		
C	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$		

	State	Logs		
A	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	
B	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$		
C	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$		

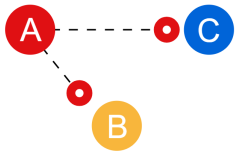
	State	Logs		
A	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	
B	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	
C	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	

	State	Logs		
A	$T:3, V:A, x:5, C:2$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	
B	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	
C	$T:3, V:A, x:3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	

	State	Logs		
A	$T:3, V:A, x:5, C:2$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	
B	$T:3, V:A, x:5, C:2$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	
C	$T:3, V:A, x:5, C:2$	$1:x \leftarrow 3$	$2:x \leftarrow 5$	

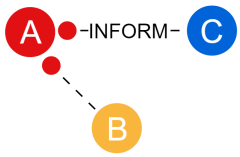
Figure 2.13: Simplified Raft server log replication where T is the current term, V is the voted for candidate, and C is the commit index. (1-3) The client's request is redirected from B to A. Then A propagates the command $x \leftarrow 5$ as log index 2. (4) both B and C replicate the log and send back a heartbeat. (5) A commits log 2, applies it to state, and sends an acknowledgment back to the client. (6) B and C update the newly committed index and apply it to state.

(1)



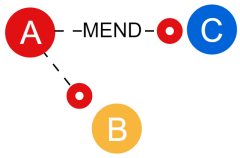
	State	Logs	
A	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
B	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	
C	$T:3, V:A, x:\varnothing, C:\varnothing$	$1:x \leftarrow 9$	

(2)



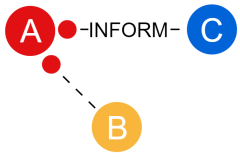
	State	Logs	
A	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
B	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
C	$T:3, V:A, x:\varnothing, C:\varnothing$	$1:x \leftarrow 9$	

(3)



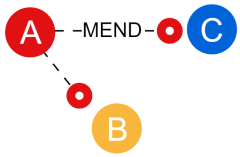
	State	Logs	
A	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
B	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
C	$T:3, V:A, x:\varnothing, C:\varnothing$	$1:x \leftarrow 9$	

(4)



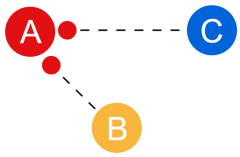
	State	Logs	
A	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
B	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
C	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	

(5)



	State	Logs	
A	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
B	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
C	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	

(6)



	State	Logs	
A	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
B	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$
C	$T:3, V:A, x:\ 3, C:1$	$1:x \leftarrow 3$	$2:x \leftarrow 5$

Figure 2.14: (1) Server C has an incorrect log, whilst server A sends out the command $x \leftarrow 5$. (2) Server C receives the command, but rejects it and informs A where they differ. (3) Server A re-sends the log from which they differ. (4) Server C corrects such entry, and informs A again. (5) Server A retries once more. (6) Server C receives the log and replicates it, and is thus up-to-date.

2.3.4 Cluster Reconfiguration (Adding, Removing, and Replacing Servers)

Cluster reconfiguration involves adding, removing, and replacing servers, or adjusting routines:

Definition 3.11: Stop and Restart & Pause and Resume

- **Stop and Restart:** Immediately halt operations, save state, reconfigure, and restart.
- **Pause and Resume:** Pause executing incoming requests and instead save them in a buffer, finish current processes, reconfigure, restart and resume.

Raft takes a two phase approach to reconfigure the cluster:

Definition 3.12: Raft Joint Consensus Reconfiguration (Part 1)

Joint consensus allows for a cluster to service requests while reconfiguring. It does this via a transition state, which is a combination of the old and new configurations (Joint consensus).

Reconfiguration: Once a leader receives a configurations request to swap from C_{old} to C_{new}, it appends $C_{old,new}$ to its logs, proceeding as follows:

- **Phase 1:**
  1. Propagate $C_{old,new}$ to all servers.
  2. Servers receiving new configurations (e.g., $C_{old,new}$), acknowledge it (**regardless** if it's committed or not).
  3. A leader crashing leads to reelections under C_{old} or $C_{old,new}$.
  4. Once $C_{old,new}$ is committed, the leader appends C_{new} to its log and replicates it.
- **Phase 2:**
  1. C_{new} is committed and live, any server not under C_{new} can be shut down.
  2. The leader who committed C_{new} becomes a follower to account for **Deletion**.

Deletion: To remove servers, they are excluded in the C_{new} configuration; **With the only exception** that leaders who do not exist in C_{new} may continue to manage the cluster until C_{new} is committed.

Addition: New Servers enter with **empty logs** and are added in the following manner:

1. The leader upon receiving the C_{new} request is notified of new server existence.
2. New servers **cannot vote**, but only receive logs (as to avoid splitting the cluster).
3. Once new servers are synchronized in logs, $C_{old,new}$ may be committed.
4. $C_{old,new}$ clusters require the majority vote from **both old and new servers**.

Though we must address the situation where servers outside of C_{new} try to cast votes:

Definition 3.13: Raft Joint Consensus Reconfiguration (Part 2)

Since Raft processes requests asynchronously, during the transition to the new configuration from $C_{old,new}$ to C_{new}, the following could happen:

- A server not yet notified of the new configuration may not know it has been removed from it. Hence, it may not receive heartbeats and attempt to start a new election.
- A server with some other configuration comes online and begins to send out RPCs to the cluster.

In any such case of unwanted votes, Raft enacts the following protocol to ensure safety:

- **Leader Liveliness:** If servers within C_{new} believe they have a leader, they ignore votes (regardless of higher order terms).
- **Majority Rule:** Even if an election occurs, C_{new} members will vote amongst themselves, for which servers outside of C_{new} will never win.

Ideally, we would like to immediately terminate servers that are removed, but this may not always be possible or practical.

2.3.5 Log Compaction & Snapshotting

In finite systems, logs cannot expand without bound. So as to clean log entries without losing state consensus, snapshotting is employed.

Definition 3.14: Raft - Log Compaction & Snapshots

Raft compacts logs via state snapshots (1.1). Servers independently take snapshots, and **are not** initiated by the leader. Snapshots employ the following protocol:

1. Trigger snapshots when logs reach a fixed size in **bytes** or time (e.g., see below tip).
2. The snapshot replaces **committed** log entries and retains entries yet to be committed.
3. The snapshot includes last committed log index, term of last index, and machine state (e.g., key-value pairs).

Tip: Simple Snapshot Initiate Protocol - One possible solution suggested by the Raft paper is to initiate the snapshot when the log reaches a certain size in bytes. Ideally a size significantly larger than the size of the expected snapshot. Since snapshots have a **fixed cost** in setup, it reduces the overall cost overtime if snapshots occur at fewer intervals.

Proof 3.1: Raft - Log Compaction & Snapshots

Leader initiated snapshots may overcomplicate its RPC and or make costly use of bandwidth. Since the leader has already vetted committed log entries, it is safe to assume that the resulting independent snapshots consisting of committed logs—are also safe. ■

Though it's not ideal to send an snapshots over the network, in some cases we might have to.

Definition 3.15: Raft Resynchronization (InstallSnapshot RPC)

There are two cases where a snapshot may be sent:

1. **Missing Entry**: If the leader has already compacted a log entry it needs to send to a server, it sends the snapshot instead.
2. **Server Recovery**: A server coming online may be severely behind, and thus, simply sending AppendEntries RPCs may take too long.

In such cases the leader will send a **InstallSnapshot RPC**. Such RPC includes the last log index. The receiver discards all its logs before the last log index, and retains the rest.

Copying the entire dataset is costly, as to avoid this, Raft recommends a lazy snapshotting approach:

Definition 3.16: Copy-on-Write (CoW)

Servers mark existing data as shared between a snapshot and live state. Only when a write (modification) touches shared data, does the server copy to preserve the snapshot.

Definition 3.17: Raft - Handling Stale Logs in Client Interactions

Raft ensures clients receive up-to-date logs from leaders via the following methods:

- **Command IDs:** Each command is assigned unique serial numbers. Each server maintains the last processed command ID and associated response for each client.
- **Syncing Leader Logs:** To avoid stale leader logs, two precautionary are taken:
  1. At the beginning of each term, the leader replicates a dummy *no-op* command log and waits for it to be committed.
  2. The leader sends responses only after exchanging heartbeats with the majority.

2.3.6 InstallSnapshot RPC Schema

Below is a summary of the Raft RPC **InstallSnapshot** for implementation:

InstallSnapshot RPC	
Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.	
Arguments	**Description**
term	Leader's term.
leaderId	Allows follower to redirect clients.
lastIncludedIndex	The snapshot replaces all entries up through and including this index.
lastIncludedTerm	Term of **lastIncludedIndex**.
offset	Byte offset where chunk is positioned in the snapshot file.
data[]	Raw bytes of the snapshot chunk, starting at **offset**.
done	**True** if this is the last chunk.
Response to Sender	
term	Current term, for leader to update itself.
Reciever Implementation	
1. Reply immediately if **term** < **currentTerm**. 2. Create new snapshot file if first chunk (**offset** = 0). 3. Write data into snapshot file at given **offset**. 4. Reply and wait for more data chunks if **done** is false. 5. Save snapshot file, discard any existing or partial snapshot with a smaller index. 6. If existing log entry has the same index and term as snapshot's last included entry, retain log entries following it and reply. 7. Discard the entire log. 8. Reset state machine using snapshot contents (and load snapshot's cluster configuration).	

Consider the below figure which illustrates a snapshot taken from a server A:

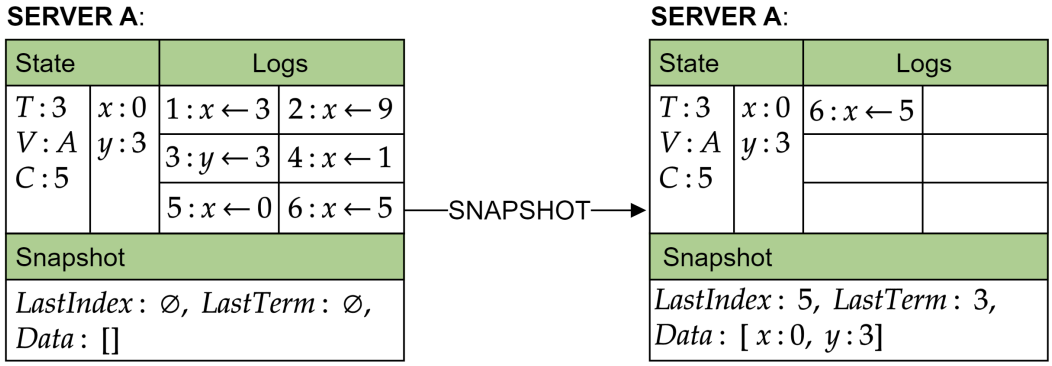


Figure 2.15: Server A begins with the initial state where, T is the current term, V is the voted for candidate, and C is the commit index. There is a separate partition for the snapshot where, $LastIndex$ is the last committed log entry's index, and $LastTerm$, the term from which $LastIndex$ resided. The arrow points towards after the snapshot, from which we see 6, the remaining log entry that wasn't committed.

2.3.7 Raft Algorithm Paper

Below is the original source with additional proofs and explanations [6]:

- https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf
- **Extended Version:** https://raft.github.io/raft.pdf

And these two websites which offer interactive visualizations of the Raft Algorithm:

- http://thesecretlivesofdata.com/raft/
- https://raft.github.io/

2.4 Failure Models

2.4.1 Defining Failures

This section discusses, what to do or how to classify when failures occur.

Definition 4.1: Failure Model

A **Failure Model** is a set of assumptions about the types of failures that can occur in a distributed system. Such failures may be **Correlated** or **Independent**. Processes that do not fail are considered **Correct**.

In particular, we are concerned with the following types of failures:

- Crash and or omission of responses, and how we might recover from them.
- Deviation protocol, arbitrarily or maliciously.

Definition 4.2: Crash-Stop Failure

A **Crash-Stop Failure** occurs when a process halts and does not resume (e.g., power failure, software crash). The failure is not explicitly detectable by other processes. **Permanent hardware failures** typically fall under category of crash-stop failures.

Definition 4.3: Fail-Stop Failure

A **Fail-Stop Failure** is a detectable crash-stop failure (e.g., timeouts, heartbeats).

Definition 4.4: Omission Failure

An **Omission Failure** can be categorized into two types:

- **Send Omission**: The process fails to send messages according to the protocol.
- **Receive Omission**: The process fails to receive messages that were sent by other processes.

In particular, The process itself may still be operational but unable to correctly communicate (e.g., network disruptions, software errors, or buffer overflows).

Definition 4.5: Crash-Recovery Failure

A **Crash-Recovery Failure** occurs when a process halts due to a crash but retains the **ability to recover** and resume execution.

- **Crash Phase**: The process halts in some way (e.g., stops sending or receiving messages).
- **Recovery Phase**: The processes may recover to the last correct state via snapshot (2.1). Certain types of memory may persist through the crash:
  - **Volatile memory** is lost during a crash (e.g, mid-execution variables).
  - **Stable storage** is retained through a crash (e.g., disk storage, assuming no disk failure).

However, if the processes crashes indefinitely, it is considered a **Crash-Stop** failure (4.2).

Definition 4.6: Byzantine Failure

A **Byzantine Failure** occurs when a process exhibits arbitrary or malicious behavior, leading to unpredictable system behavior.

- **Arbitrary Behavior**: The process deviates from the expected protocol, such as:
  - Sending corrupted or inconsistent messages to different nodes.
  - Updating its state in an unintended or unpredictable manner.
- **Malicious Behavior**: The process actively attempts to disrupt the system, such as:
  - Exploiting protocol vulnerabilities to manipulate outcomes (e.g., double-spending in a blockchain).

In short, these failures occur due to **bugs** (unintentional) or **attacks** (intentional).

Tip: The term **Byzantine** in computer science comes from the *Byzantine Generals Problem*, introduced by Leslie Lamport in 1982. It describes a scenario where generals must coordinate an attack but cannot trust all messengers—some may be traitors sending conflicting information.

The name *Byzantine* is inspired by the Byzantine Empire, which was historically known for its complex and often deceptive political intrigues. While the term is widely used in distributed systems, some argue it unfairly portrays Byzantine history.

In computing, a **Byzantine failure** refers to a system component acting unpredictably, whether due to bugs, faults, or malicious intent, making consensus difficult.

2.4.2 Failures Model Hierarchy

Here we compare failure models as extensions of each other, though we will omit **fail-stop** as it is more of a detection mechanism. To quickly recap:

- **Crash-Stop**: Process halts and cannot resume (undetectable).
- **Omission**: Process fails to properly communicate.
- **Crash-Recovery**: Process halts but can recover and resume.
- **Byzantine**: Process exhibits arbitrary or malicious behavior.

Theorem 4.1: Failure Model Hierarchy

The failure models can be arranged in a hierarchy, where each model is an extension of the previous one. The hierarchy is as follows:

$$\textbf{Crash-Stop} \subset \textbf{Omission} \subset \textbf{Crash-Recovery} \subset \textbf{Byzantine}$$

In particular,

- **Crash-Stop** $\subset$ **Omission**: A crash-stop is a full crash rather than a partial communication failure.
- **Omission** $\subset$ **Crash-Recovery**: During recovering the last correct state, volatile memory lost may exhibit omission-like behavior. Meaning some messages are lost due to "**amnesia**."
- **Crash-Recovery** $\subset$ **Byzantine**: as a process may recover and exhibit arbitrary or malicious behavior. Moreover, a Byzantine failure may be **any type of failure**.

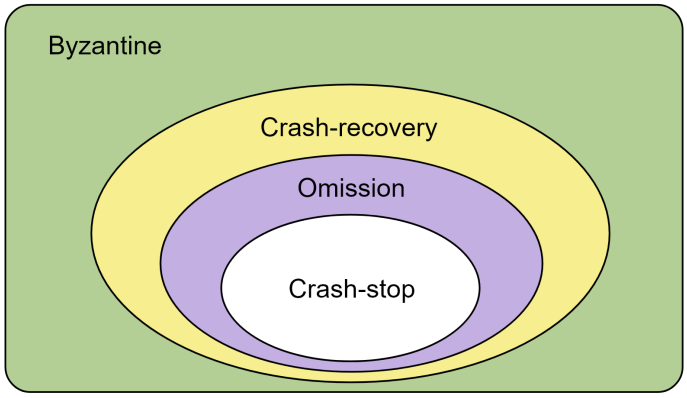


Figure 2.16: Failure Hierarchy depicting: Crash-stop $\subset$ Omission $\subset$ Crash-Recovery $\subset$ Byzantine

2.5 Consistency Models

2.5.1 Introduction

Moving on from consensus models, which handle agreement on data values, we now shift our focus to the ordering of operations and their validity:

Definition 5.1: Consistency Model

Within a distributed system, a consistency model acts as a contract between systems on valid operation ordering (e.g., reads or writes) on shared data within the network.

These models are critical for ensuring that a system behaves as expected when replicating data across multiple nodes. [8]

Definition 5.2: Global Total Order

A Global Total Order is a sequence of operations that is agreed upon by all nodes in a distributed system. This means that such sequence given some client inputs, could reproduce the same state on a single machine.

Now we begin to list some common consistency models:

Definition 5.3: Strong Consistency

The client observes that all nodes *appear* to agree on the order of execution. This means all node reads of shared data are identical (Global Total Order of operations).

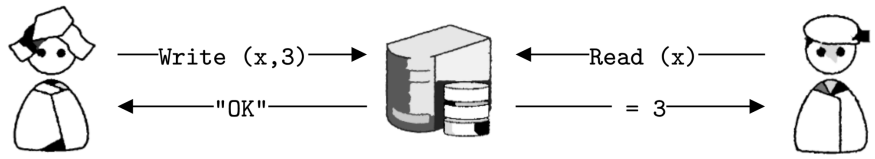
Definition 5.4: Weak Consistency

The client **temporarily** observes that nodes disagree on shared data values at some point in time (no Global Total Order).

Theorem 5.1: Strong Consistency vs Weak Consistency

Even though strong consistency is desirable, it can be costly on the network and difficult to implement. Weak consistency is easier to implement and may provide better performance, but at the cost of data integrity and difficulty in debugging interactions.

Consider this example interaction, which interaction has the weakest consistency model?



Find The Weakest Consistency

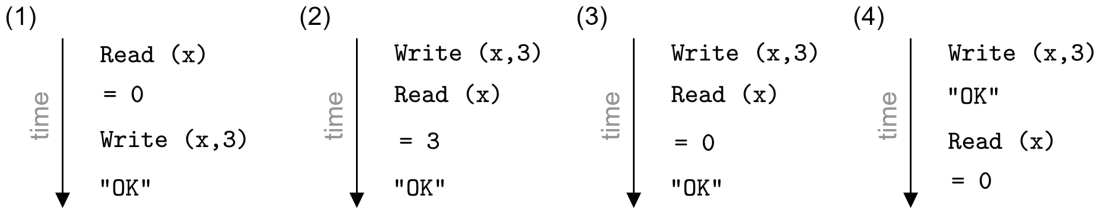


Figure 2.17: (1) Strong Consistency, as the read x could be 0, and the write responds with "OK". (2) This is okay, as our write of 3 is read. The "OK" does come a little late, but its order isn't bizarre. (3) Still okay, as the "OK" comes after the read of 0, meaning the write could not have happened yet. (4) The weakest, as our read completely disregards our acknowledged write of 3.

2.5.2 Strong Consistency Models: Linearizability & Sequential Consistency

We discuss two strong consistency models, Linearizability and Sequential Consistency:

Definition 5.5: Linearizability

Replicas produce a Global Total Order, which preserves real-time ordering of events. Moreover, every read must return the value of the **most completed** write. In particular:

- If operation A completes before B, then $A \to B$ in real-time.
- If tasks A and B overlap, there is no real-time ordering.

Theorem 5.2: Raft and Linearizability

Raft's leader-commit design provides Linearizability, **except** in specific scenarios such as:

- A committed log entry is lost within the majority of the cluster.
- The newly elected leader somehow bypasses the Leader Completeness property.

In all other cases, Raft is considered to have a strong consistency model.

Consider the following examples and determine whether it is linearizable:

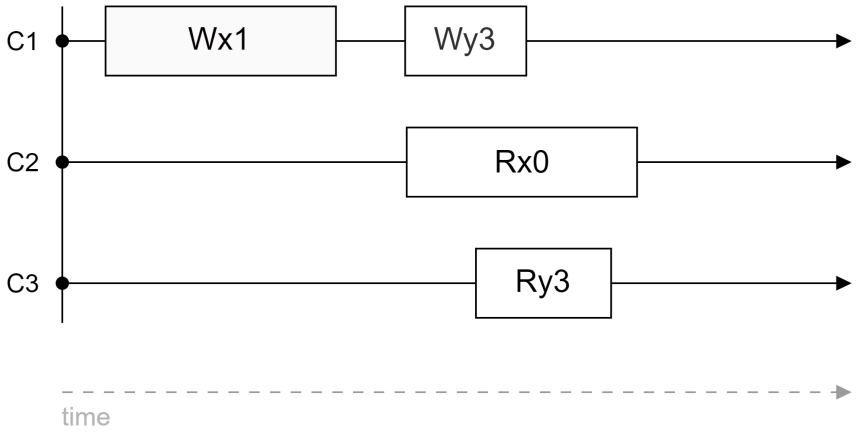


Figure 2.18: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx0$ reads, "0 read from x." This system is not linearizable because the read $Rx0$ should have returned 1, in respect to real-time.

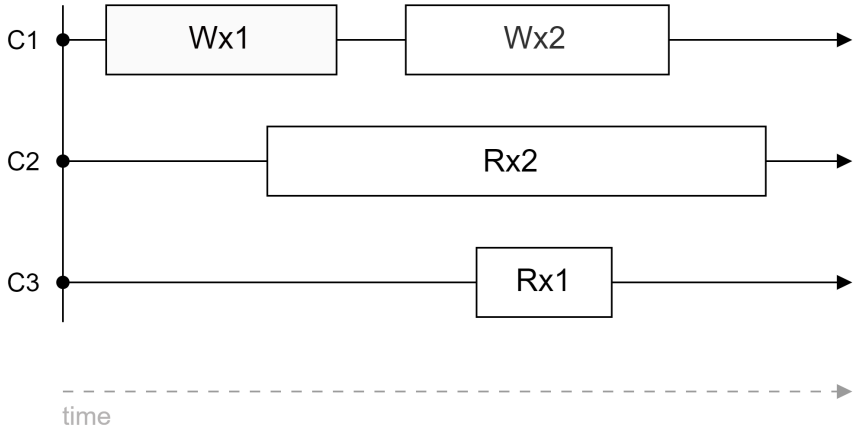


Figure 2.19: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x." This system is linearizable. This is because each box represents a period of time when the action could take place. Therefore, Wx1 and Rx1 can happen, while still having enough time for Wx2 and Rx2 to occur.

The **next page** includes an elaboration of the above.

Here we elaborate on Figure (2.19):

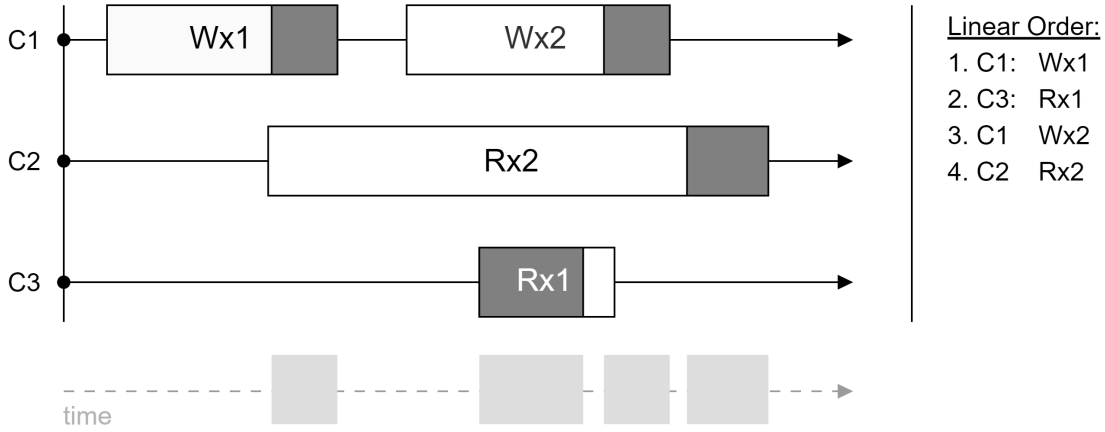


Figure 2.20: The gray boxes represent a mutex on the shared data, x. Here we see the order of operations, Wx1, Rx1, Wx2, and Rx2, from which **no overlaps occur** and logically make sense.

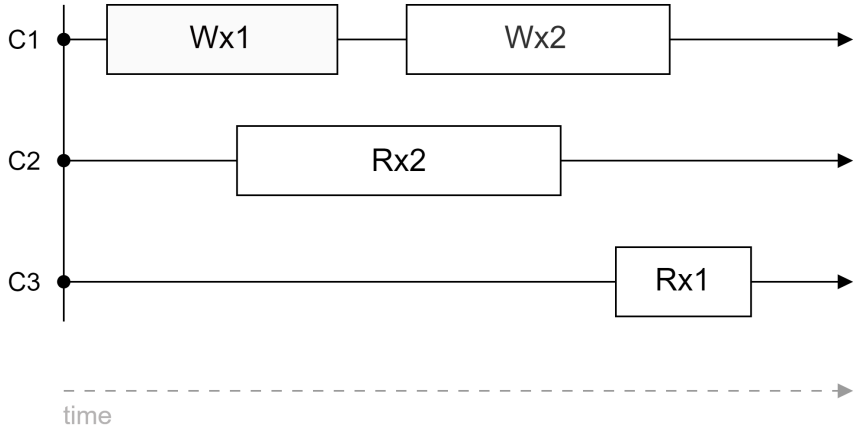


Figure 2.21: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x." This system is not linearizable. The Rx1 is not possible as it's too far disjointed from Wx1 after being overwritten by Wx2.

We've talked about another replication method before that is also linearizable:

Theorem 5.3: Chain Replication and Linearizability

Chain Replication is linearizable. The duration of a write operation extends until the tail, from which an acknowledgment is sent. Reads also adhere to reading the last completed write, and is thus—a strong consistency model.

The next model is Sequential Consistency:

Definition 5.6: Sequential Consistency

Weaker than Linearizability. All replicas execute all operations in **some** Global Total Order. Each client **observes the same order** once such order is agreed upon. Therefore a single machine may replicate the system if given such order. In particular:

- If a system process issues A before B, then $A \to B$ in the Global Total Order.
- If A and B happen on different processes, there is no real-time ordering (concurrent).

Consider the following examples and determine whether it is sequentially consistent:

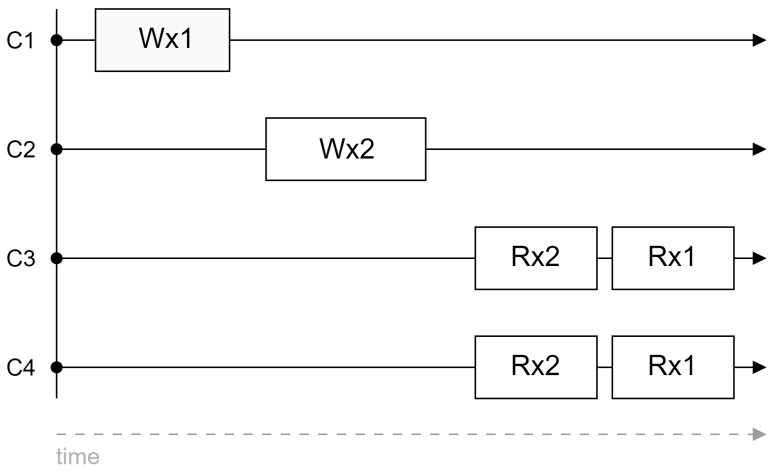


Figure 2.22: A distributed system with client C1, C2, C3, C4 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx0$ reads, "0 read from x." This figure may or may not be sequentially consistent.

Next page includes an elaboration of the above.

We elaborate on the previous example:

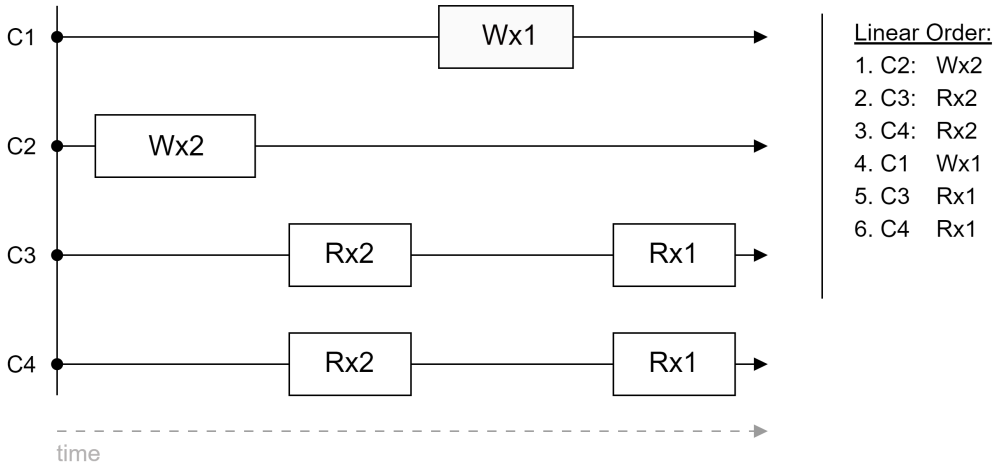


Figure 2.23: A distributed system with client C1, C2, C3, C4 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x.". This figure is sequentially consistent as there exists some Global Total Order that can be agreed upon. Here listed, Wx2, Rx2, Rx2, Wx1, Rx1, Rx1.

And lastly, consider the following example:

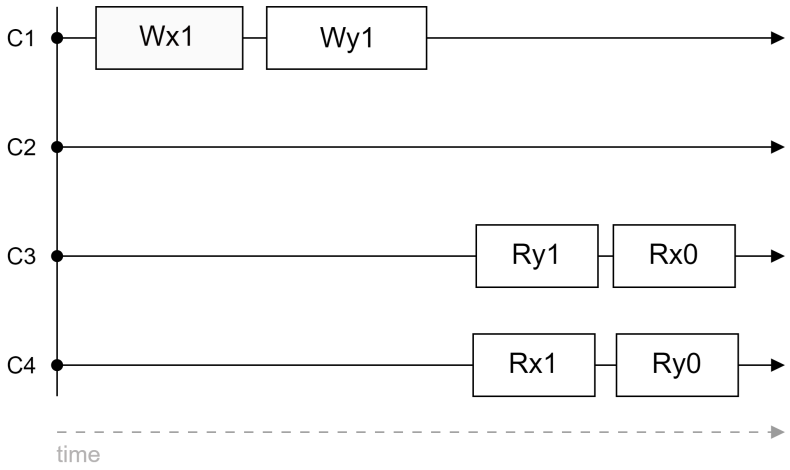


Figure 2.24: A distributed system with client C1, C2, C3, C4 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x.". This figure is not sequentially consistent as there is no Global Total Order that can be agreed upon. Here Rx0 cannot happen after Ry1 because of the relation Wx1 $\rightarrow$ Wy1.

Consider the following theorem:

Theorem 5.4: Linearizability vs Sequential Consistency

If a system is linearizable, it is also sequentially consistent. As, adhering to real-time order naturally satisfies sequential consistency. **However**, the reverse is not true. In particular:

- **Linearizability:** Relies on real-time.
- **Sequential Consistency:** Relies on program order.

2.5.3 Handling Shared Data via Mutex: Release & Lazy-release Consistency

Consider the following **Weak** methods of handling shared data:

Definition 5.7: Release Consistency vs. Lazy-release Consistency

These methods require explicit use of locks to propagate updates:
Release Consistency: Push updates to all nodes after releasing the lock.
Lazy-release Consistency: Push updates to the next node who acquires the same lock. *Though this may provide less stress to the system, if the client does not lock data, it may become stale (weak consistency). Alternatively, if the client locks all data (strong consistency) though it may be costly.* **These both are weak consistency models.**

Determine whether the following system is release or lazy-release consistent:

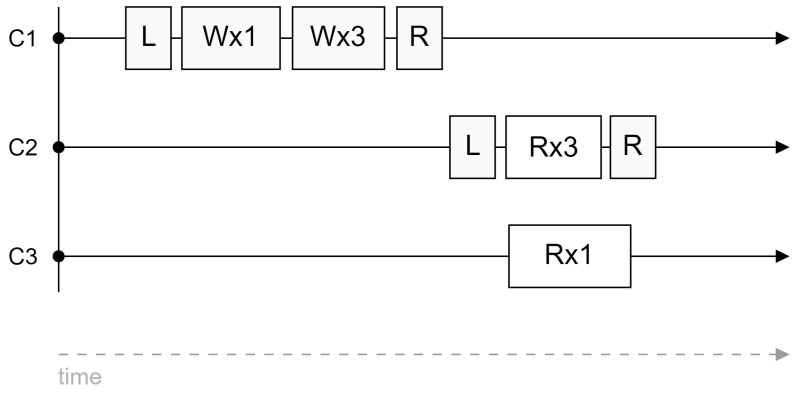


Figure 2.25: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x." With L (lock) and R (Release). This is Lazy-release consistent, as C3's read of x wasn't updated to 3 in absence of a lock.

2.5.4 Weak Consistency Models: Causal & Eventual Consistency

We now discuss two weak consistency models, Causal Consistency and Eventual Consistency:

Definition 5.8: Causal Consistency

Weaker form of Sequential Consistency. That the Global Total Order of operations adhere logically to their causal dependencies. In particular:

- If A causes B, then $A \rightarrow B$ should be in the Global Total Order.

I.e., Causal Consistency is just Sequential Consistency, differing slightly in that Clients in Causal Consistency may observe different Global Total Orders.

For example, only one of these sentences makes causal sense:

(1)

1. Alice: Lunch?
2. Bob: Yes.
3. Carl: No.

(2)

1. Bob: Yes.
2. Alice: Lunch?
3. Carl: No.

(3)

1. Alice: Lunch?
2. Carl: No.
3. Bob: Yes.

Here, statements (1) and (3) are causally consistent, as the responses are in order of the question. Take that same intuition for this next example and determine whether the below system is causally consistent:

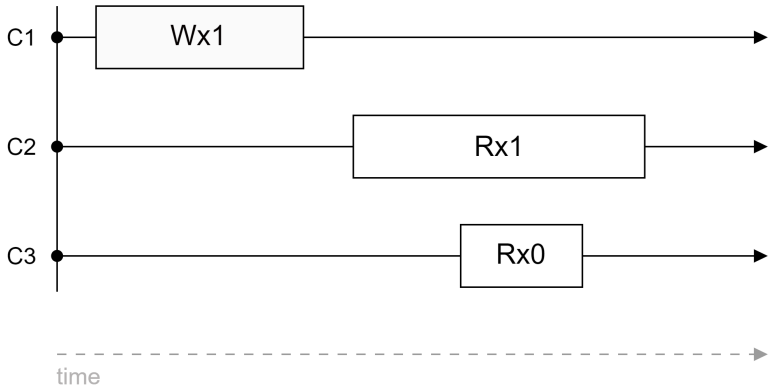


Figure 2.26: A distributed system with client C1, C2, and C3 interactions. Where $Wx1$ reads, "Write 1 to x" and $Rx1$ reads, "1 read from x." This system is causally consistent as the Global Total Order of operations adhere to their causal dependencies. Here, Wx1 $\rightarrow$ Rx1, where Rx0 must come before Wx1.

Consider the following example and determine whether it is causally consistent:

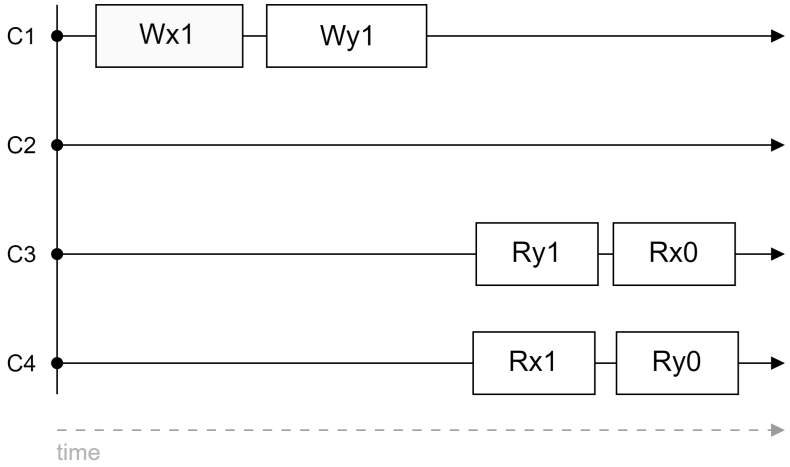


Figure 2.27: This A distributed system with client C1, C2, C3, and C4 interactions. This is not causally consistent, as $Wx1 \rightarrow Wy1$, but $Ry1 \rightarrow Rx0$ does not adhere to this relationship.

Definition 5.9: Eventual Consistency

Weaker than Causal Consistency. Given there are no new writes, replicas will **eventually** agree on the same value after some period of time.

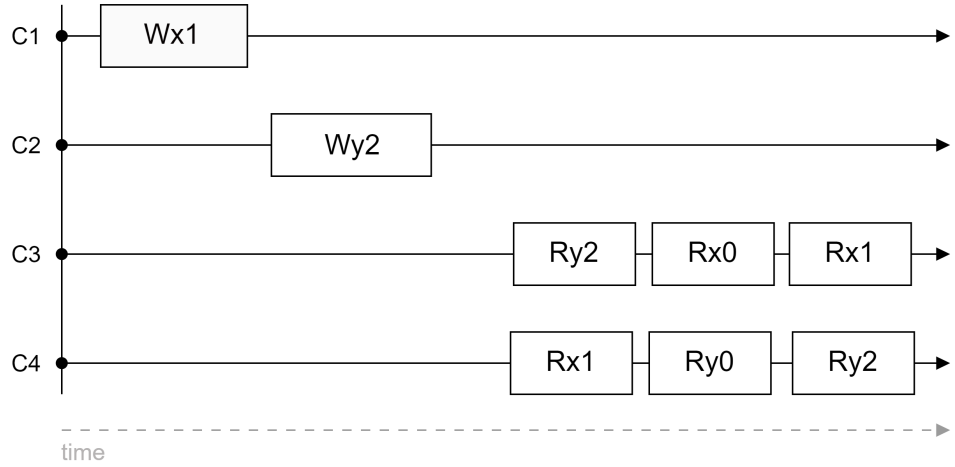


Figure 2.28: This system is eventually consistent as it eventually agrees on x and y.

Causal Consistency implies two properties:

Theorem 5.5: Causal Consistency → FIFO & RYW

Causal Consistency implies FIFO (First-In-First-Out) and RYW (Read-Your-Writes):

- **FIFO:** All writes are read in the order they were issued.
- **RYW:** If the client writes to x, then their next read of x must return the value they just previously wrote.

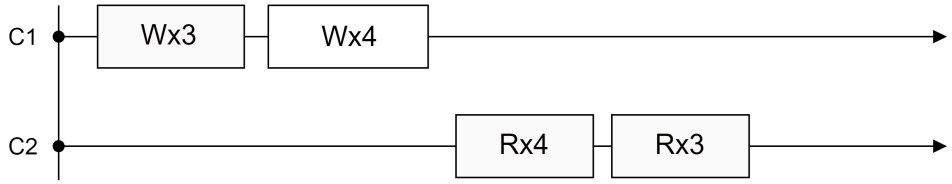


Figure 2.29: This system is not FIFO nor Causally Consistent. Writes **must be read in monotonic order**. Hence, Rx3 must come before Rx4.

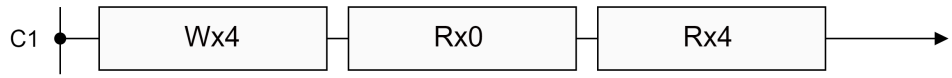


Figure 2.30: This system is not RYW nor Causally Consistent. As the next read should be Rx4 alone. The order, Rx0 → Wx4 → Rx4, satisfies RYW and is Causally Consistent.

Tell whether the following is linearizeable, sequential, causal, and or read your writes.

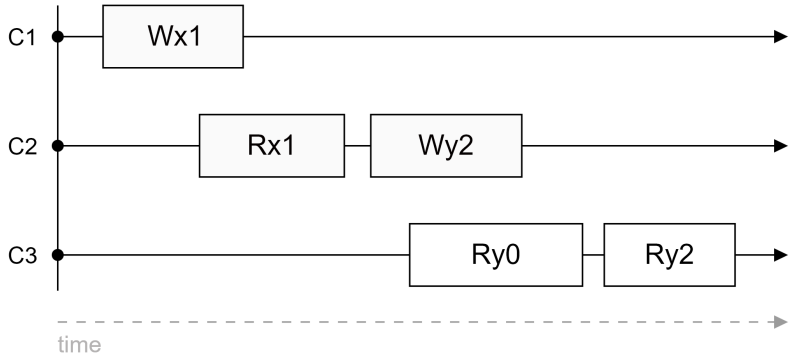


Figure 2.31: This system is linearizable, sequential, causal, and RYW.

Recall that linearizability deals with real-time ordering of locks.

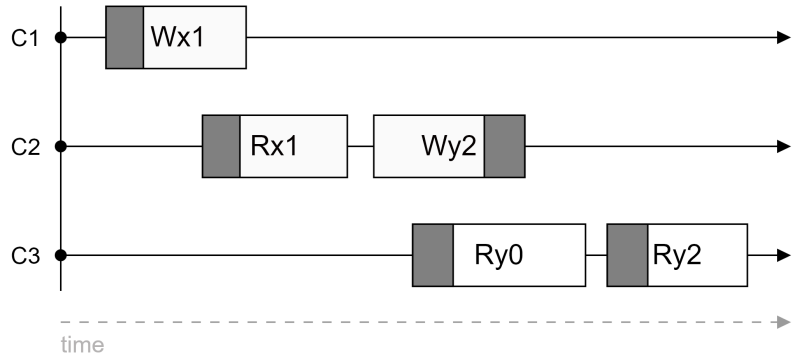


Figure 2.32: Here, locks are claimed in non conflicting order, which follows a real-time order.

To emphasize:

Theorem 5.6: Consistency Model Implications

The following implications hold among consistency models:

$$\text{linearizability} \rightarrow \text{sequential} \rightarrow \text{causal} \rightarrow (\text{RYW/FIFO})$$

2.6 Transactions and Concurrency Control

2.6.1 Optimistic Concurrency Control (OCC)

Say the backbone of our stock trading application is a distributed database. The system may conduct complicated stock trades based on server stock prices.

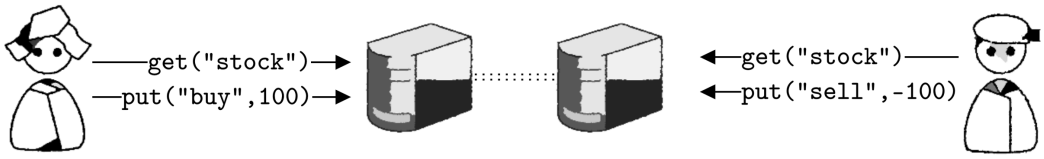


Figure 2.33: Two users checking the stock prices and making a trade based on such information.

It is critical that the stock price is consistent through all servers, and more important that if any trade fails, the system can recover to a consistent state.

Definition 6.1: Transaction

A **transaction** is a sequence of operations that are treated as a single unit of work. A transaction must satisfy the **ACID** properties:

- **Atomicity**: A transaction is either fully completed or dropped entirely.
- **Consistency**: A transaction takes the database from one valid state to another.
- **Isolation**: Transactions must be isolated from each other.
- **Durability**: Once a transaction is committed, it remains so even after system failure.

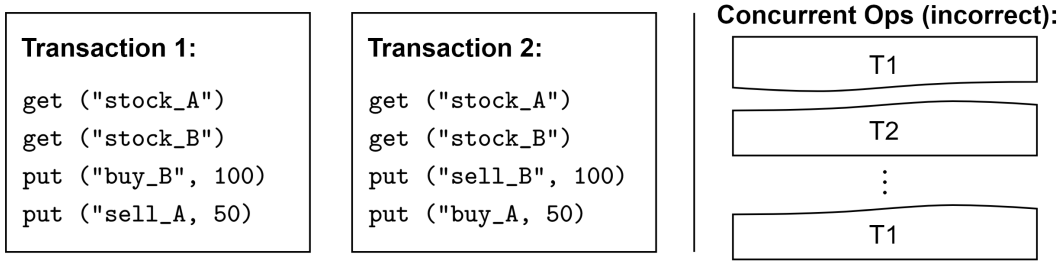


Figure 2.34: Two transactions which whose operations are interleaved.

Interleaving transactions **violates the isolation property**. This is problematic in Figure (2.34) as T2's (transaction 2) operations may depend on the server state before T1's transaction. Additionally partially completed transactions leave the system in an inconsistent state, violating **atomicity** (e.g., T1's "buy_B" fails, but "sell_A" succeeds).

We discuss another consistency model which will help us in this settings:

Definition 6.2: Serializability

Serializability is a strong consistency model that ensures the outcome of concurrent transactions is the same as if they were executed in some sequential (serial) order.

This differs from **linearizability**, which focuses on the real-time ordering of individual operations. Serializability instead concerns the logical order of **entire transactions**, independent of their timing.

However, **strict-serializability** does care about real-time ordering in addition to the logical order of transactions. This implies linearizability, but not vice versa.

We consider the following model to help us preform transactions:

Definition 6.3: Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control (OCC) assumes conflicts are rare and proceeds without locking. It follows four main steps:

- **Prepare**: The system reads the transaction request and creates a backup or temporary copy of the state.
- **Modify/Validate**: The transaction modifies the temporary state. Then The system checks whether the transaction is **serializable**.
- **Commit/Rollback**: If valid, commit; Otherwise, abort transaction and rollback to previous state.

This only solves **isolation**, as it does **not** guarantee atomicity.

Definition 6.4: Transaction Coordinator and Database Servers

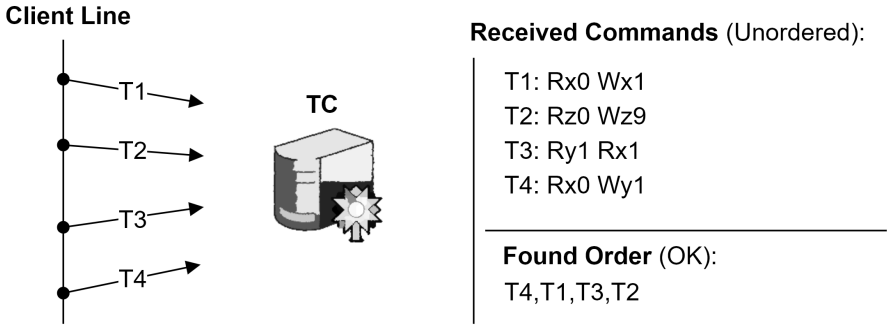
OCC maintains two necessary components:

- **Transaction Coordinator (TC)**: The validation server responsible for checking whether a transaction is **serializable**. It receives requests from clients and responds with either:
  - `OK`: if the transaction is serializable,
  - `ABORT`: if it conflicts with prior transactions.
- **Database Servers (DB)**: Receives transaction operations, executing it on local state. Then, on **OK**—commit state, on **ABORT**—rollback to the previous state.

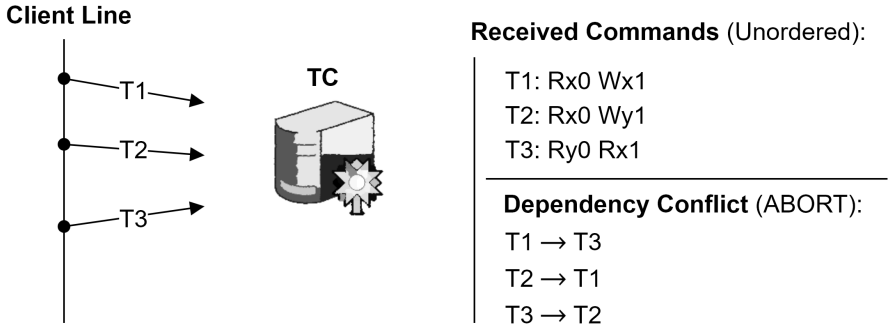
We consider one model, which where multiple clients interact with one TC.

Example 6.1: Centralized OCC

Consider these two examples with a single TC and multiple clients on the network line:



Here, clients on the line send transactions T1–T4 to the TC. The TC then checks the transactions for serializability. In this case an order is found (T4,T1,T3,T2), which the TC communicates to the DBs to commit.



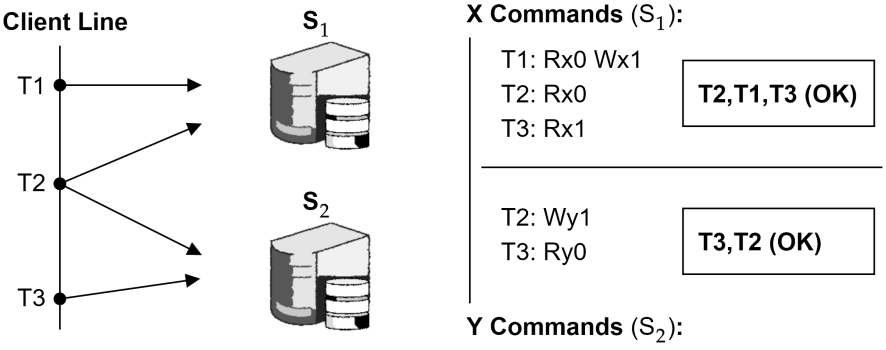
Here, transaction requests, T1, T2, and T3, do not have a serial order. As we build, T1 → T3 (T1 then T3) makes logical sense. Then T2 → T3, giving us the order T2 → T1 → T3; However, it appears T3 must come before T2. This causes a cycle (T2 → T1 → T3 → T2). Hence, the TC must must abort all transactions. ■

Tip: If familiar with **Directed Acyclic Graphs (DAG)**, one can think of the transactions as nodes and the edges as the dependencies between them. If the graph is a DAG, then there is some serial order (OK). If not, then there is a cycle, so we must abort.

Though we run into an issue with server specific data, the TC must resolve execution order.

Example 6.2: Distributed OCC

Consider two servers responsible for different parts of our system data.

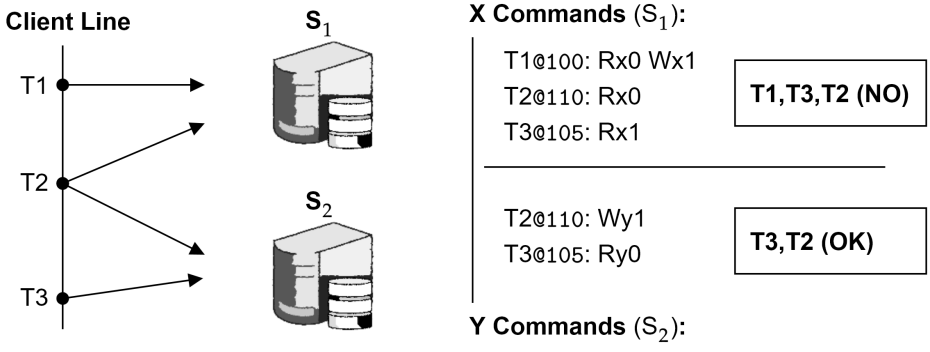


Problem: The TC picks **different serial orders** for each server during transactions. ■

Theorem 6.1: Timestamping Distributed OCC

Timestamping is a method where each transaction is assigned a unique timestamp (ID), which aids order agreement between TCs. The downside: **unnecessary aborts.**

Example 6.3: Timestamping Distributed OCC



Timestamps (@#) only serve as IDs. Here S_2 OKs the order (T3,T2). The TC sees this, and enforces the order for S_1, but it isn't able to serialize. Hence, the TC aborts (NO). ■

2.6.2 Two-Phase Commit (2PC)

We introduce another method to help us with this problem, though it **does not ensure isolation**:

Definition 6.5: Two-Phase Commit (2PC)

Two-Phase Commit (2PC) ensures **atomicity**. The client sends the transactions to the DBs (participants). There after, the client tells the TC start the commit process, involving two phases:

- **Prepare Phase**: The coordinator sends a prepare request to all DBs; Each respond:
  - `YES`: if it can commit the transaction.
  - `NO`: if it cannot commit the transaction.
- **Commit Phase**: If all voted `YES`, the coordinator sends a `COMMIT` request to all DBs. If any participant voted `NO`, the coordinator sends an `ABORT`. The TC then responds to the client with the final outcome of the transaction.

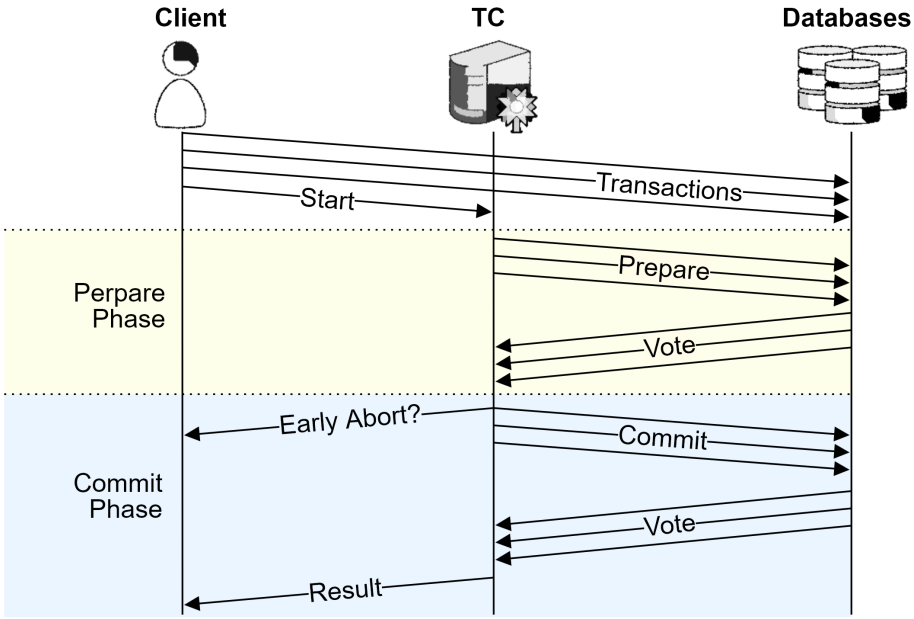


Figure 2.35: Two-Phase Commit (2PC) process. The client sets up the transaction with the TC and DBs. Then, the TC starts the commit process, starting with the prepare phase, and then ends with the commit phase.

Though there is a pitfall with this method:

Definition 6.6: 2PC Blockout

In most cases if there's a timeout the TC or DBs will abort the transaction. However, if the TC times out after a DB has voted `YES`, the DB is left in an uncertain state, and must **wait** for the TC to respond.

Preparation Phase	
Case	**Action**
TC timeout for yes/no vote	**Abort** — transaction coordinator did not receive votes in time.
DB timeout for prepare	**Abort** — database did not receive prepare request in time.
Commit Phase	
Case	**Action**
DB timeout for commit/abort and DB voted NO	**Abort** — since DB already voted NO, it's safe to abort.
DB timeout for commit/abort and DB voted YES	**Block** — DB must wait for TC decision to preserve atomicity.

Definition 6.7: 2PC Persistent & Volatile State

In the Two-Phase Commit (2PC) protocol, both the Transaction Coordinator (TC) and Database (DB) participants must persist critical information to recover correctly after a crash.

- **Database (DB):**
  - Must persist the result of any vote (`YES` or `NO`).
  - If the DB voted `YES` and then crashes, upon recovery it must contact the TC to learn the final decision (`COMMIT` or `ABORT`).
- **Transaction Coordinator (TC):**
  - Must persist the result of any vote and the final decision (`COMMIT` or `ABORT`).
  - If the TC crashes, it must resume the commit protocol from where it left off.

2.6.3 Three-Phase Commit (3PC)

2PC's main problem was **availability**. We can fix this by adding an additional phase:

Definition 6.8: Three-Phase Commit (3PC)

Three-Phase Commit (3PC) is an extension of 2PC that adds a third phase to avoid blocking in case of failures:

- **CanCommit Phase**: The coordinator sends a `CANCOMMIT` request to all participants. Each participant responds with either `YES` or `NO`.
- **PreCommit Phase**: If all participants respond with `YES`, the coordinator sends a `PRECOMMIT` request to all participants. Participants prepare to commit sending back an acknowledgment (`ACK`).
- **DoCommit Phase**: If all participants `ACK` the precommit, the coordinator sends a `COMMIT` request. Otherwise, it sends an `ABORT` request.

Theorem 6.2: 3PC Non-Blocking Nature & Self-Resolution

The reason this fixes the blocking issue in 2PC lies within the `PRECOMMIT` phase. In regular 2PC, if a DB votes `YES` it must wait for the TC to respond as it is uncertain whether another DB has committed or not.

In 3PC, such DB need not worry, as no other DB can commit until the TC sends a `PRECOMMIT` request. Henceforth, DBs can safely either terminate or resolve between themselves that if everyone else has the `PRECOMMIT` request, they can commit.

Theorem 6.3: 3PC Persistent & Volatile State

Both the TC and DBs may crash at any time, for which they must persist the state of the transactions. Though it is fully possible for the TC to only persist the original transaction request, and instead query **all** DBs for their state to recover the transaction.

Theorem 6.4: 3PC Tradeoffs

- (−) If the network becomes partitioned, inconsistencies may arise within self-resolution.
- (−) Adds another round trip to the commit process (3 in total).
- (+) Removes the blocking issue of 2PC, as the TC can always recover from a crash.

Below is a diagram of the 3PC process:

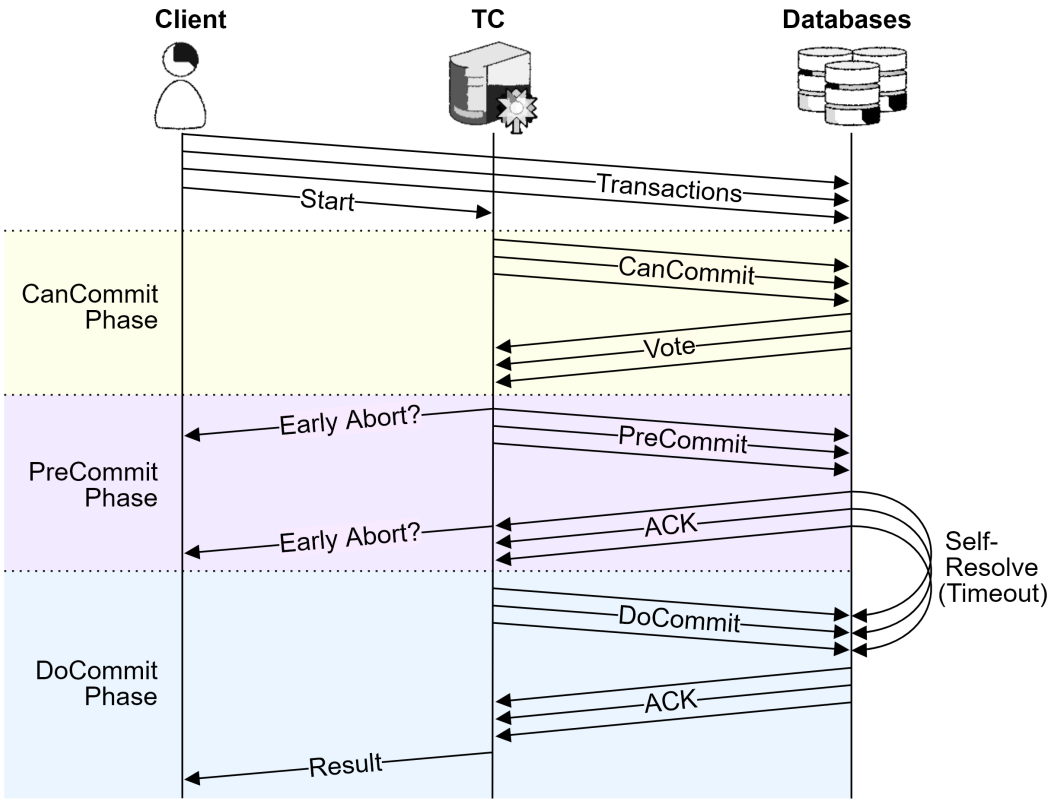


Figure 2.36: Three-Phase Commit (3PC) protocol message flow. The transaction progresses through three phases: **CanCommit**, where participants vote; **PreCommit**, where participants acknowledge readiness to commit; and **DoCommit**, where the final decision is executed. The TC may decide to abort if it recieved a `NO` vote, or failed to `ACK` during `PreCommit`. In the event of coordinator failure, participants may invoke **Self-Resolve** after a timeout, based on whether they received a `PreCommit` message, ensuring non-blocking recovery. **Note:** The Databases do not communicate with the client after resolution, though this could depend on the implementation.

We briefly go over OCC's opposition:

Definition 6.9: Pessimistic Concurrency Control (PCC)

Pessimistic Concurrency Control assumes conflicts are likely and prevents them by acquiring locks before any data access. It then preforms actions directly on shared data, ensuring **serializability and isolation**. The locks are released after the transaction is completed.

2.7 Virtual Memory*

2.7.1 Problem Space

One may skip this section if they are already familiar with pages, or have reached the section detailing TLBs. The rest is offered for completeness. Virtual memory solves three problems [3]:

- **Not enough memory, Memory fragmentation, and Security**

Definition 7.1: Not Enough Memory

Back then, computer memory was expensive, and many computers had very little memory (e.g., 4–1 GiB or even less). CPUs could only support up to 4 GiB of memory, as CPUs were 32-bit (2^{32} addresses $= 2^{32} bytes = 4GiB$). On the other hand, 64-bit CPUs can support up to 2^{64} addresses = 16 million TB of memory.

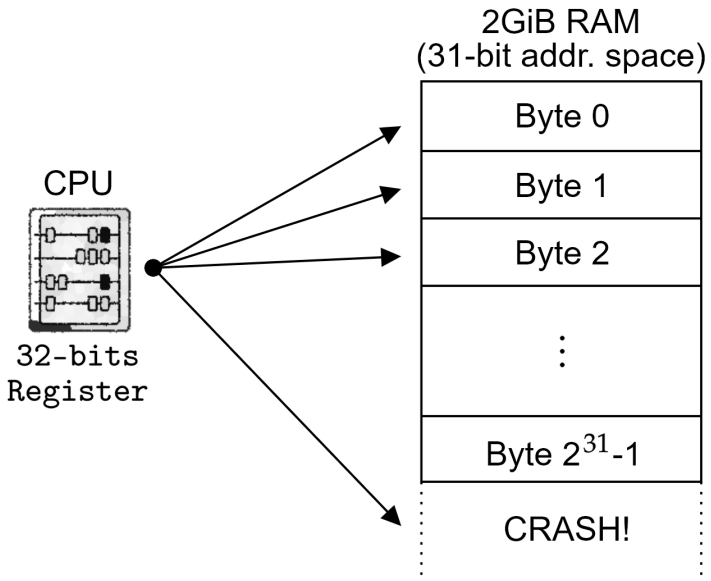


Figure 2.37: A 32-bit CPU accessing 2 GiBs of RAM, where a crash happens when trying to access beyond the 31-bit address space.

The next problem deals with multiple processes allocating and deallocating memory:

Definition 7.2: Memory Fragmentation

Memory can be thought of as a big array, where each cell is a resource a program can use. We want memory usage to be contiguous (i.e., no gaps or holes). So say we have an array

$$[O, O, O, O]$$

Where O represents free space in our array, each cell 1 GiB of space. If we have processes A and B take 1 and 2 GiBs respectively, we might have a memory layout of:

$$[A, B, B, O,]$$

If we then free process A, we might have a memory layout of:

$$[O, B, B, O]$$

Now, if another process C needs 2 GiBs of memory, it will not be able to find a contiguous space of 2 GiBs, even though we have 2 GiBs of free space. This is called **memory fragmentation**.

Now finally we have the problem of protecting memory from other processes:

Definition 7.3: Memory Security

In a multi-process system, processes may have collisions when trying to access the same memory space. For example, if process A is a weather service and process B is some finance service, we don't want the weather service to overwrite the same memory space where the finance service is storing critical data. This is called **memory security**.

So in theory we want to give each process it s own portion of memory, to solve overlapping access:

Definition 7.4: Virtual Memory

To solve process memory collisions, we give each process its own fictional view of memory, called **virtual memory**. Though for this to work, each virtual view is mapped to an actual place in the original memory we call **physical memory**.

Virtual and physical addresses are the cells spaces themselves.

Consider the figure below showing how virtual memory works in theory:

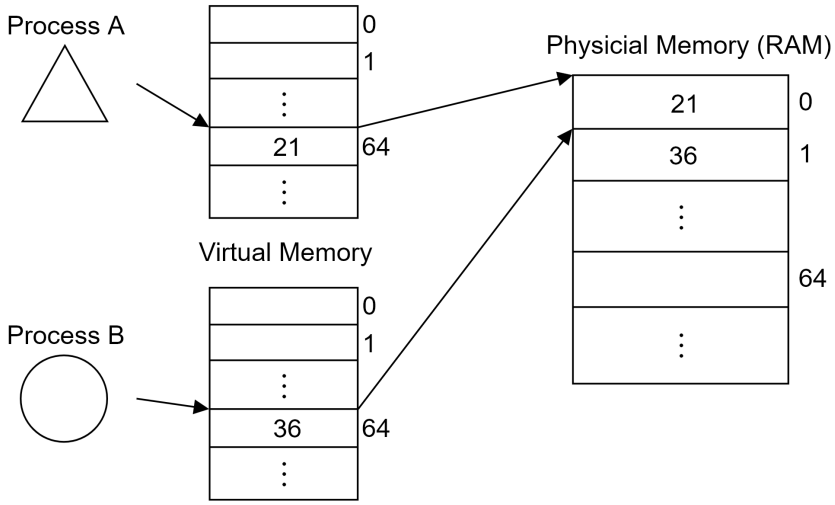


Figure 2.38: Processes A and B write to memory cell 64 in their view of memory, but in reality they map to different physical memory cells (0 and 1 respectively).

A quick aside:

Theorem 7.1: Physical Memory the CPU Accesses

In reality the CPU can access the physical memory of many other devices (e.g., hard drives, SSDs, etc.). In addition, the OS takes up some of the physical memory as well. The rest is left to programs to use. The program allocated space is the memory we refer to going forward as the **physical memory**.

Virtual memory solves the three problems we mentioned before:

Definition 7.5: Virtual Memory & Not Enough Memory (Swapping)

The physical memory can be much smaller than what a program thinks it has in virtual memory. When a program tries to access memory it does not have, the OS will **swap** physical memory to external storage to free up space. This means, while a program is not using a portion of memory at a given time, the OS can swap it in and out depending on system needs.

Memory that is swapped out is called **swap-memory**. Every time a we try to access such absent memory in our mappings, it's called a **page fault** (more on this later).

Definition 7.6: Virtual Memory & Fragmentation

Virtual memory allows programs to think they have contiguous memory. This simplifies their memory management, while in reality the OS manages the split memory mappings.

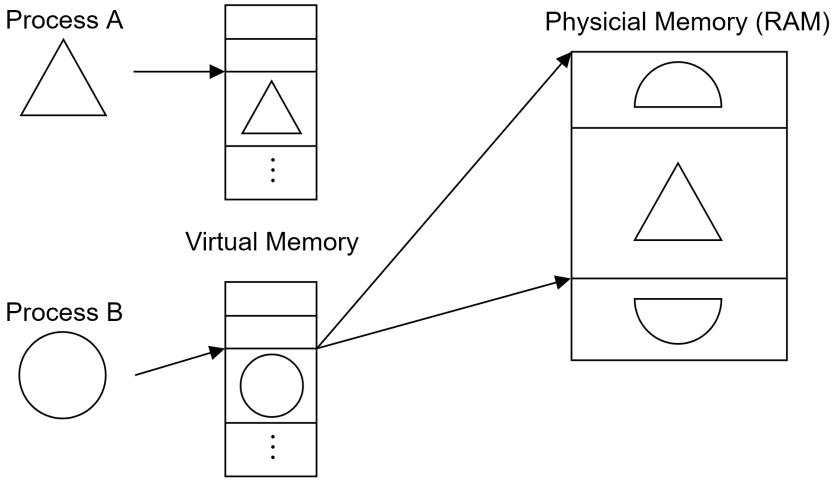


Figure 2.39: Here two processes A and B are using the same physical memory space. Both think they have contiguous memory, but in reality, B is split into two parts in the physical memory.

Definition 7.7: Virtual Memory & Security

Memory cells are collision safe as memory mappings are not shared in physical memory; However, this would be inefficient if say A and B depend on a secondary process C. In this case, A and B may share the same mapping to C's memory.

2.7.2 Virtual Memory Implementation (Page Tables)

We discuss the mapping mechanism further in linking virtual to physical memory.

Definition 7.8: Page Tables

The OS keeps a **page table**, where each entry is a mapping of a virtual memory cell to a physical one, called a **Page Table Entry (PTE)**. CPUs work with words (32 bits = 4 bytes), so the page table has one entry for every word (address) in the virtual memory space.

We make the following observation:

Theorem 7.2: Theoretical Page Table Space Complexity

If our CPU 32-bits, then there are 2^{32} addresses. CPUs speak in words, hence we'd need 2^{30} words, and thus ≈ 1 billion PTEs (4 GiB per table). This is too much memory to practically implement for each process.

We solve the above problem, via the following strategy:

Definition 7.9: Memory Chunking (Pages)

Instead of mapping each address to a PTE—which would be too large. We chunk the memory into **pages** reducing the number of PTEs needed. Typically, page sizes are some power of 2, most commonly 4 KiB (4096 bytes), meaning 1024 words per page. This reduces our page table requirement from 4 GiB to 4 MiB (1 Million PTEs). Despite moving 4 KiBs of data at a time, this works well in practice, as adjacent memory is often accessed together.

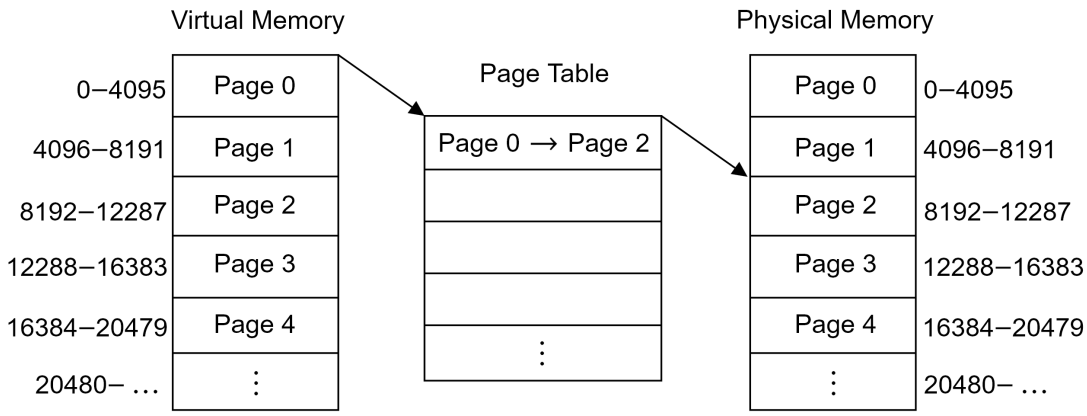


Figure 2.40: A 4 MiB page table, each PTE 4 KiB (4096 bytes, 1024 words), showing mappings.

One must ask themselves:

- What is the relationship of how the Page Numbers are separated.
- Can we determine the page number given a specific address?
- How might we convert a virtual to a physical address given some address?

We discuss the following on the next page.

Example 7.1: Converting Virtual to Physical Memory (intuition)

We may find conversions from virtual to physical memory via the following formula:

$$\text{PA} = \underbrace{(\text{VA} \ \% \ \text{Page Size})}_{\text{offset}} + \underbrace{(\text{Page Size} \cdot \text{Physical Page Number})}_{\text{Physical Page starting index}}$$

Where % is modulo and Page Number $= \left\lfloor \frac{\text{Addr.}}{\text{Page Size}} \right\rfloor$. **However,** this is not the most efficient way, and is strictly for educational/intuition building purposes.

Given the Figure (2.40), $104 \to 8296$ from $104 + 4096 * 2 = 8296$. ■

Theorem 7.3: Converting Virtual to Physical Memory

Addresses are binary numbers, typically represents as hexadecimal (base 16) numbers.

The first 12 bits of an address is the **offset**. The remaining higher-order bits yields the **page number**. To find the physical address, take the last 12 bits of the address, index the higher-order bits into the page table, and append the offset to the physical page number.

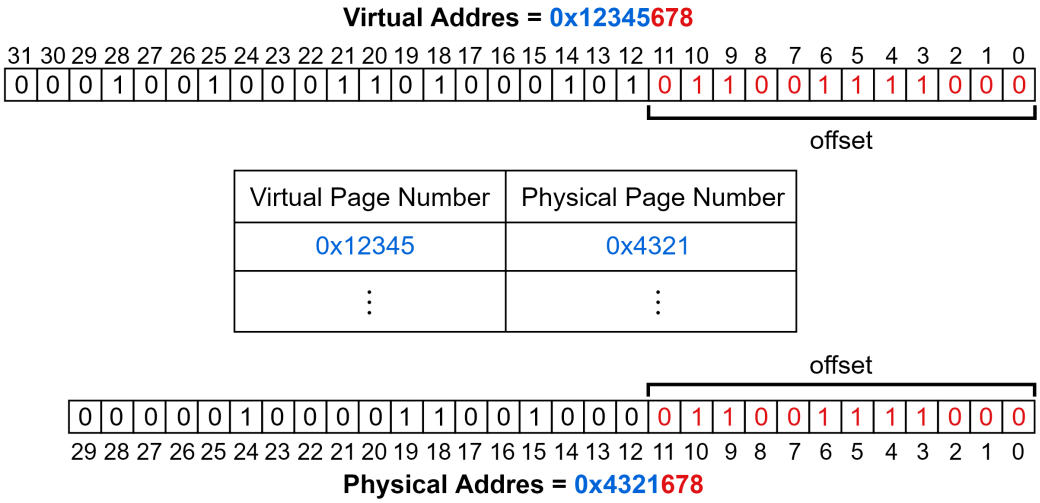


Figure 2.41: The conversion of 0x123456 (32-bit, 4 GiB) → 0x4321678 (30-bit, 1 GiB). Recall that if a accessing a page that is not in memory, causes a page fault, from which the OS with begin to swap memory with external storage devices.

2.7.3 Page Faults & Translation Lookaside Buffer (TLB)

So far we have discussed how the mapping system of virtual memory works, but now we pivot to how the OS handles swapping data in and out of memory on page faults.

Definition 7.10: Swapping on Page Faults

A **page fault** occurs when a program tries to access a page that does not have a mapping in the page table. The OS then picks the **least recently used (LRU)** page in the page table, and swaps it out to external storage. A page is **dirty** if it has been written to. In such case, we write back its contents to external storage before swapping. If the page is not dirty, we may discard it on swap.

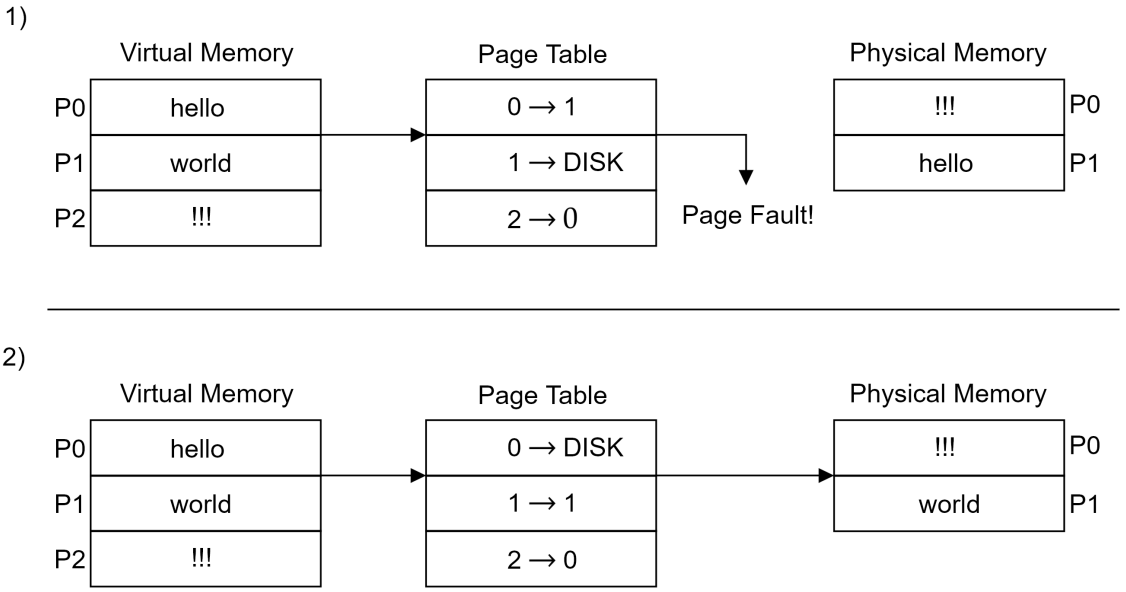


Figure 2.42: 1) A page fault occurs when trying to access page 1. 2) The OS has swapped out page 0 to external storage (DISK), and now page 1 can be accessed. Note: This diagram is for illustrative purposes, virtual memory does not contain data, only addresses binding to physical memory.

Definition 7.11: Direct Memory Access (DMA)

Page faults are expensive, as swapping data with I/O devices may take some time. Modern CPUs have a **Direct Memory Access (DMA)** module, which allows I/O devices to access memory directly, while the CPU completes other tasks.

Still our routine of mapping is expensive in it of itself.

Definition 7.12: Translation Lookaside Buffer (TLB)

The mapping process includes three steps:

1. Index the page table: Access Memory (RAM).
2. Convert Addresses: Computation.
3. Interact: Access Memory (RAM).

To speed this up, we create a small cache of the most recent Page Table Lookups, called the **Translation Lookaside Buffer (TLB)**. If a page in not in the TLB, we must preform the translation then load it into the TLB for future use.

Every successful TLB lookup is called a **hit**, while a failed lookup is called a **miss**. The **hit time** and **miss time** are the time it takes to access the TLB and page table respectively (measured in CPU clock cycles). The **hit rate** is the ratio of hits to the total number of lookups.

Modern CPU architectures usually have two TLBs, one for instructions (**ITLB**) and one for data (**DTLB**).

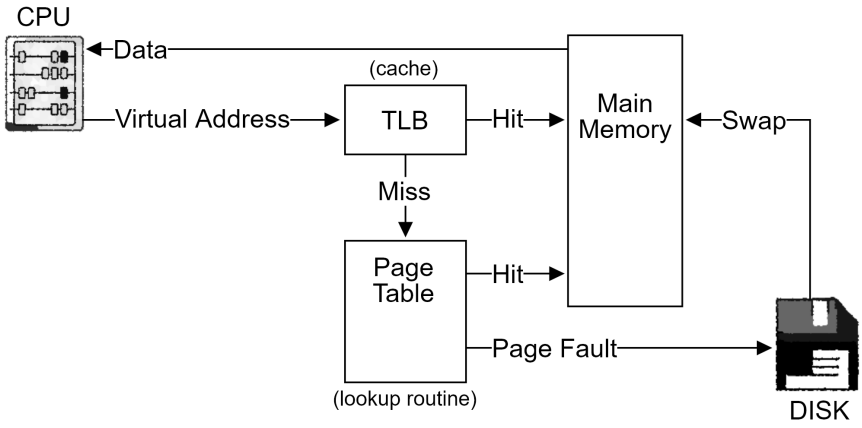


Figure 2.43: Demonstrating how a TLB lookup effects the memory access flow. CPU attempts to access memory providing the virtual address to the TLB. (Happy path) A hit occurs, memory is accessed and passed to the CPU. (Ok path) A TLB miss occurs, causing a page table lookup. A page table hit occurs, the mapping is cached and the memory is accessed. (Sad path) both a TLB and page table miss occurs, causing a page fault. The OS must decide which page to swap out. After swapping, the addressed is cached and the memory is accessed.

So far what we've been covering actually resides within the CPU architecture:

Definition 7.13: Memory Management Unit (MMU)

The **Memory Management Unit (MMU)** is a hardware component that manages the mapping of virtual to physical memory. It is responsible for translating virtual addresses to physical addresses, and it also handles page faults and TLB lookups.

2.7.4 Multi-level Page Tables

Let's define the problem space of multi-level page tables [1]:

Theorem 7.4: Single Level Page Table Cost

In a 32-bit system, each page table requires 4 MiB of memory. If we had 100 programs running, that's easily 400 MiB of memory.

Additionally, if we move page tables to disk (external storage), we lose any reference we had to them in an attempt to free up memory. So we need to keep some reference oracle in memory.

This highlights the need for a more efficient way

Definition 7.14: Multi-level Page Tables

Recall that in a 32-bit system, we have 4 GiB of memory. In single level page tables, we chunk our addresses into 4 KiBs (4096 bytes), and load all of them at once. That's

$$4 \text{ KiB} \cdot 1024 \text{ PTEs } = 4096 \text{ KiB} = 4 \text{ MiB}$$

We wish to offload data without losing references. To do this we allocate a single chunk of memory (4 KiB) to serve as an reference oracle, called the **page directory** or **1^{st} level page table**. Each entry in the page directory points to other chunks from which we are safe to load and unload out of memory. These referenced chunks are called the **2^{nd} level page tables**. Each entry in the 2^{nd} level page table points to a physical page.

Definition 7.15: Converting Virtual to Physical Memory (Multi-level)

In multi-level page tables, the first 12 bits are the **offset**, the next 10 bits are the index into the **second-level page table**, and the last 10 bits are the index into the **first-level page directory**. These tables are treated as arrays: indexing the first table yields the address of the second table, and indexing the second table yields the physical page number.

Recall, in single-level page tables the higher 20 bits are the page number and the index, which yields the physical page number. Multi-level breaks away from this, and instead as illustrated below:

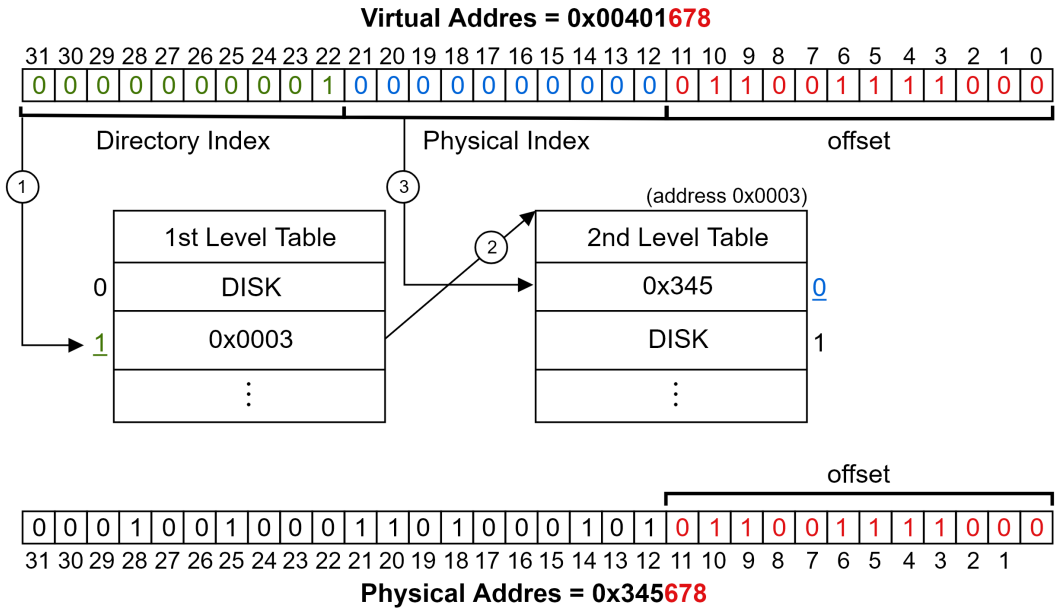


Figure 2.44: A multi-level page table, where the first level is a page directory, and the second level is a page table which points to physical pages.

Definition 7.16: Multi-level Page Scaling (64-bit)

Increasing the number of bits allows us to scale the number of levels in our page table. In 64-bit architectures using 4-level paging (e.g., x86-64), only the lower 48 bits are used. The virtual address is divided into five parts:

- 12 bits for the page offset
- 9 bits for the 4^{th}-level page table (PT)
- 9 bits for the 3^{rd}-level page directory (PD)
- 9 bits for the 2^{nd}-level page directory pointer table (PDPT)
- 9 bits for the 1^{st}-level page map level 4 (PML4)

Modern versions of Windows and Linux support 5 levels, which allows for a maximum of 129 PiB, while 4 gives us 256 TiB.

Finally we talk about the performance of our lookups:

Definition 7.17: Effective Memory Access Time (EMAT)

Effective Memory Access Time (EMAT) accounts for the time it takes to access memory in the presence of a Translation Lookaside Buffer (TLB), multi-level paging, and potential page faults. Given:

- t = TLB access time; m = Memory access time
- S = Page fault service time; n = Number of page levels
- h = TLB hit rate; p = Page hit rate (1 - page fault rate)

The EMAT is computed as:

$$\text{EMAT} = h(t+m) + (1-h)\,(t + p(n \cdot m) + (1-p)S)$$

Essentially [4],

```
EMAT=
TLB hit*(TLB access time + Memory access time)
+ TLB Miss*(TLB access time + PageHit*[n * memory access time]
+ PageMiss*PageFaultServiceTime)
```

Example 7.2: Three-Level Paging System

Suppose the system has:

- $n = 3$ page levels
- $t = 5$ ns (TLB lookup)
- $m = 100$ ns (memory access time)
- $\alpha = 0.80$ (TLB hit rate)

Then the EMAT is:

$$\begin{aligned}\text{EMAT} &= (5+100)\cdot 0.80 + (5 + 3\cdot 100 + 100)\cdot(1-0.80)\\ &= 105\cdot 0.80 + 405\cdot 0.20 = 84 + 81 = \boxed{165 \text{ ns}}\end{aligned}$$

For a better TLB hit rate $\alpha = 0.98$:

$$\text{EMAT} = 105\cdot 0.98 + 405\cdot 0.02 = 102.9 + 8.1 = \boxed{111 \text{ ns}} \qquad [2]$$

■

2.8 Distributed Shared Memory

To speed up computation on a single machine, we use multiple cores to run OS threads in parallel. Consider the following example:

```
    for (i := 0; i< n; i++) {
        go work(i,results);
    }
```

Here our goal is to run some function `work` on a perhaps large data set of n size. Go easily abstracts this away with the `go` keyword.

Now, what if we want to run this on a distributed system?

```
    package main
    import (
        "net/rpc"
    )
    type Args struct{}
    type WorkServer int64
    func (t *WorkServer) DoWork(args *Args, reply *int64) error {
        // Fill reply pointer to send the data back
        work(args.data, *reply);
        return nil
    }
```

We have to decide now on a system of communication:

- How do we interact with sending and receiving data (conflicts, failures, etc.)
- How should our coordinator dispatch the work?
- What RPCs should we include in our API?

To solve this problem, we reuse a hardware primitive in the previous Section (2.7).

Theorem 8.1: Multithread Communication – Virtual Memory

A multithread process can still communicate between threads by utilizing the same virtual memory space, we call this **shared memory**.

The goal is to bring this primitive to the distributed system level.

We clearly define our problem-space:

Definition 8.1: Distributed Shared Memory

A **Distributed Shared Memory (DSM)** system is a system that allows multiple processes on different machines to access a shared memory space as if it were local. I.e,. create the illusion of a single shared memory across multiple machines to enable multithreaded programs in a distrusted setting.

Applications are built on top of a DSM system who abstracts away consensus and communication. DSM systems connect to the underlying hardware support for shared memory (virtual address translation) to coordinate. Page faults are redirected to the DSM system to resolve.

Consistency Model: Strong consistency, and linearizability. **Every** read operation is up-to-date, reflecting the most recent write, if any.

The next solution will be our first draft from which we shall iterate on:

Definition 8.2: DSM – Page Ownership & Swap Protocol (Draft)

Given a system of M_i machines, the Virtual Address Space (VAS) is evenly divided amongst them. I.e., each machine has some partition of pages it owns. On a local machine, the page table will operate as normal; Though, upon a page fault, instead of going to the disk, we redirect to the DSM system. From there we request the missing page over the network.

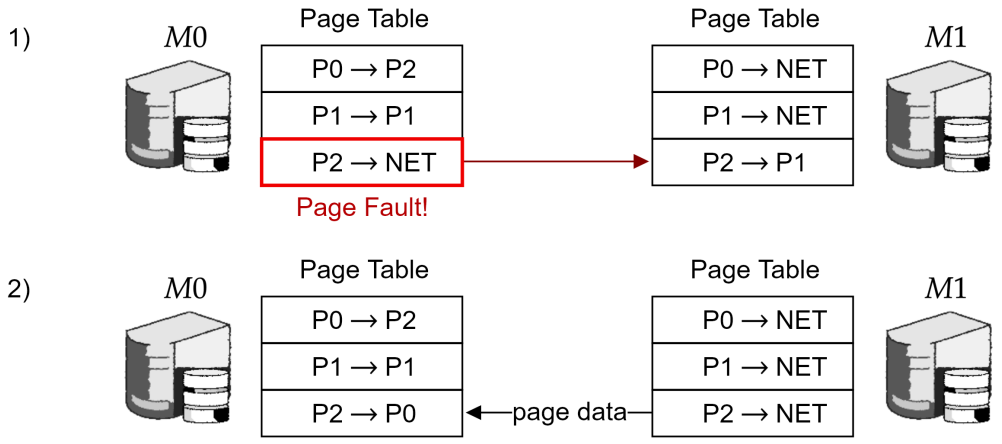


Figure 2.45: 1) System M_0 tries to access P_2 in virtual memory but incurs a page fault, prompting a request to the DSM system. The DSM resolves that M_1 owns the desired page. 2) The DSM performs a network swap, sending the actual page data so M_0 can load it into physical memory.

Though this is a good start, it's expensive:

Theorem 8.2: Sending Pages Over the Network & False Sharing

Pages are often large (4KB), which is expensive to send. If two machines M_0 and M_1 access the same page, but modify different data points (e.g., two different variables), the whole page needs to be sent even though the data is not shared. This is called **false sharing**.

A particular DSM system called **TreadMarks** solves this problem:

Definition 8.3: TreadMarks – Page Ownership & Swap Protocol

Given a system of M_i machines, the VAS is evenly divided amongst them. When page faults occur, the DSM sends **diffs** (change history) over the network instead of the entire page.

This allows for all M_i to start with **RO** (read-only) access on all pages. When M_j wishes to modify a page, M_j notifies all other M_i to invalidate their copies of the page. M_j then gets **RW** (read-write) access to the page (at most one writeable copy).

Each write creates a new V_i version of the page (i, increases monotonically). Where V_i contains the latest changes, and V_{i-1} a snapshot before the changes.

When M_i page faults, the DSM system requests the page from owner M_j, notifying them of the last version M_i saw. The owner M_j locks said page and creates a diff between both M_i and M_j's versions, sending it over the network. Both M_j and M_i revert to RO access.

Consistency Model: Weak consistency as per lazy-release style of diff sharing.

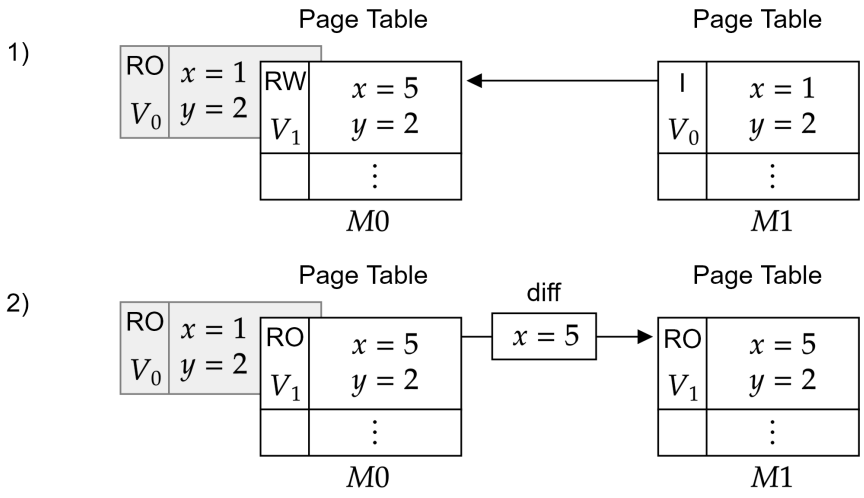


Figure 2.46: 1) $M1$ requests a page from $M2$. 2) $M2$ sends the diff between V_0 and V_1 to $M1$.

To ensure that multiple RW access occurs on the same data we send locks over the network. We briefly discussed these methods in Subsection (2.5.3) concerning release consistency.

Definition 8.4: TreadMarks – Lazy-release

TreadMarks utilizes lazy-release consistency to protect against data-races. The programmer explicitly locks and unlocks shared data (e.g., mutexes on variables). Any page that is dirtied (modified) the dsm generates a diff for. Per lazy-release consistency, nodes only require diffs when trying to access data from such pages.

Though this is a good start, it loses the causal relationship between the data.

Example 8.1: Using Vector Clocks to Capture Causal Consistency

Consider the following example at initialization `x := 0; y := &x; var z *int`:

```
1.  // -- Machine M0 -- initial VC = [0,0,0]

    Lock(A)             // acquire lock A
    x = 7               // write to page A
    Unlock(A)           // increments VC[0] -> VC = [1,0,0]; diffA@[1,0,0]

    Lock(B)             // acquire lock B
    y = &x              // write to page B (stores pointer to x)
    Unlock(B)           // increments VC[0] _> VC = [2,0,0]; diffB@[2,0,0]
```

```
2.  // -- Machine M1 -- initial VC = [0,0,0]

    Lock(B)             // pulls diffs for B newer than [0,0,0]:
                        // gets diffA@[1,0,0] and diffB@[2,0,0]
                        // updates VC to [2,0,0]
    Lock(C)             // acquire lock C
    z = y               // write to page C (stores the pointer y)
    Unlock(C)           // increments VC[1] -> VC = [2,1,0]; diffC@[2,1,0]
    Unlock(B)           // no writes, but would propagate nothing new
```

```
3.  // -- Machine M2 -- initial VC = [0,0,0]

    Lock(C)             // pulls diffs for C newer than [0,0,0]:
                        // diffA@[1,0,0], diffB@[2,0,0], then diffC@[2,1,0]
    print(*z)           // prints 7 as opposed to 0
    Unlock(C)
```

Without vector clocks, M_3 would have only seen M_1's changes to C (`z = y`), missing that that A and B were modified before. Therefore M_3 would have printed 0 instead of 7. ■

Hence, we need to employ vector clocks to ensure causality. Let's define it:

Definition 8.5: TreadMarks – Causal Consistency

TreadMarks utilizes vector clocks to ensure causal consistency between page updates. Each index of the vector clock corresponds to each machine in the system. Every update monotonically increases M_i's index in the vector clock.

For instance, say there are two machines M_0 and M_1 with pages A and B respectively, resulting in a vector clock of $[0, 0]$. If M_0 modifies page A then B, it would increment its index once for A's diff and then twice for B's diff, yielding $[2, 0]$.

Once M_1 reads page B, it requests the lock from M_0. Along with the lock acquisition comes the diffs for A and B. Then M_1 applies every diff from M_0's vector clock that is greater than its own, resulting in $[2, 0]$. Only when M_1 modifies page B, does its local vector clock increase to $[2, 1]$ (while M_0's remains at $[2, 0]$).

In summary we need the following components in our RPC API:

Definition 8.6: TreadMarks – Whole Component Summary

We need the following components in our RPC API:

- **Versioning Control**: Every write to a page creates a new version with a trailing diff.
- **Causal Consistency**: Versioning history for each machine is stored in a vector clock.
- **Locking**: Every page is locked before modification, with a lazy-release system to bypass notifying the entire cluster.

Consistency Model: Weak consistency – Lazy-release + Causal Consistency.

We degraded our consistency model from strong to weak consistency to save on performance. It is completely possible to read stale data in this system; though, it is guaranteed that the data is consistent with the causal relationship of the data.

2.9 Sharding Data & Consistent Hashing

Say we have a 1 TB dataset with only 5 servers each with 200 GB of storage.

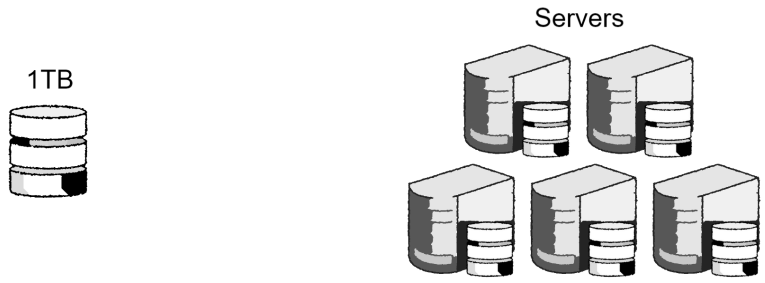


Figure 2.47: 1 TB dataset with 5 servers each with some storage capacity.

Once method is to split up our dataset into 5 partitions, and distribute them across the servers.

Definition 9.1: Sharding

Sharding is the process of splitting up a dataset into smaller more manageable pieces called **shards**. Each shard is stored on a different server, allowing for parallel processing.

In addition, to strengthen fault tolerance, replication could be used to store multiple copies of a single shard on different servers.

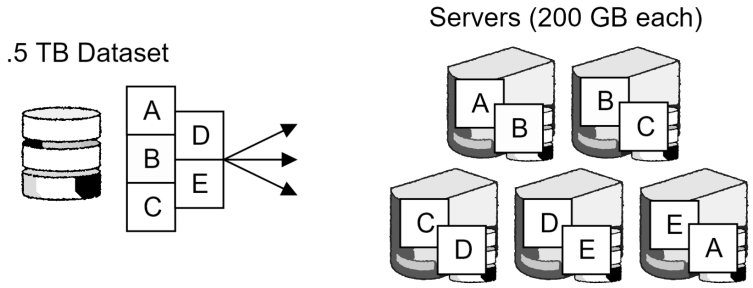


Figure 2.48: Sharding a dataset of .5 TB into 5 partitions, each part replicated twice and distributed evenly across 5 servers.

As shown above, shards could be replicated and housed with other shards on the same server. For instance, this could help in the case that two pieces of data are frequently accessed together.

Now we discuss how to assign shards to servers. There are two naive methods:

- **Randomized**: Though statistically balanced, it requires a lookup procedure to find shards.
- **Alphabetical**: Too deterministic, though we skip the lookup procedure.

Instead we could combine these ideas and use a **hash function** to map the data to a server:

Definition 9.2: Hash Function

A **hash function** is a function that takes an input (or **key**) and returns a fixed-size string of bytes. The output is typically a **digest** that is unique to each unique input. Depending on the hash algorithm, the output may be a number or a string of characters, or even produce collisions (two distinct inputs producing the same output).

Ideally the hash function should be:

- **Deterministic**: The same input should always produce the same output.
- **Uniform**: The output should be uniformly distributed across the range of possible outputs.
- **Fast**: The hash function should be fast to compute.
- **Collision Resistant**: It should be hard to find clashing inputs that produce the same output.

Though this seems fine, it may be costly to deal with migrations:

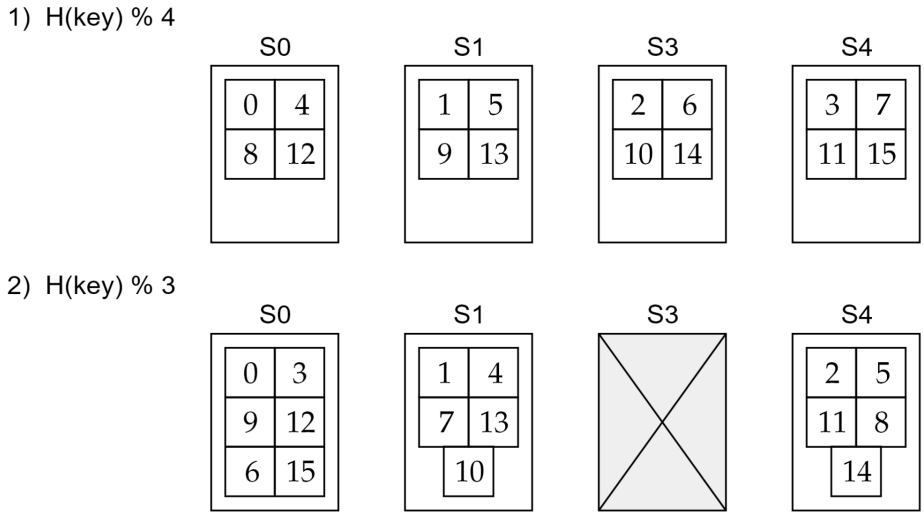


Figure 2.49: Given a hash function H and a shard ID key, where $H(key)$ returns some server ID – 1) There are 4 servers utilizing the hash function, using the size of servers to modulo overflow. 2) Server 3 goes down, forcing a rehash of the data (unnecessary migration).

To fix this migration problem, we build off the idea of the wrapping modulo overflow behavior, as well as to allow severs to handle multiple hash values:

Definition 9.3: Consistent Hashing (Part 1)

Consistent Hashing is a technique used to distribute data across a cluster of servers in a way that minimizes the amount of data that needs to be moved when servers are added or removed.

Here the hash function H maps keys to an m-bit value, which provides a hash-space of 2^m. I.e., we have 2^m possible hash values ($\{0, 1, \ldots, 2^m - 1\}$), which forms a ring, allowing us to wrap around the hash space.

Then servers S_i often called **virtual nodes** or **virtual replicas**, are evenly assigned a portion of the ring to cover. Any hash that falls within this range is assigned to the server. If a server S_i goes down, its range is passed to the next server S_{i+1} (without having to rehash the data).

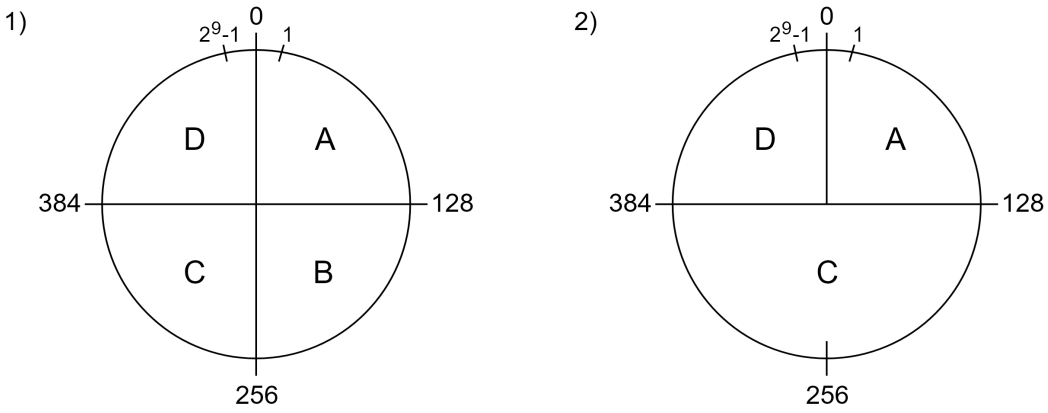


Figure 2.50: A consistent hashing ring of 9-bit values (512 possible values). The ring includes 4 servers A, B, C, D. 2) Server B goes down, transferring its range to C.

In Figure (2.50) in part 2, the data is **no longer evenly distributed** across servers after a lost server. Likewise, the same problem occurs when adding a server. We mitigate this via the following:

Definition 9.4: Consistent Hashing (Part 2)

To achieve an even distribution of virtual nodes, we in essence take server S_i's range and split it into k slices, distributing them evenly across the ring.

In particular, given n servers, and a ring of size 2^m, we create k virtual nodes to each server, requiring k different hash functions for server assignment. Typically, $k \approx \log_2(2^m) = m$. [7]

Below we illustrate how just a basic implementation of this improves our load balancing:

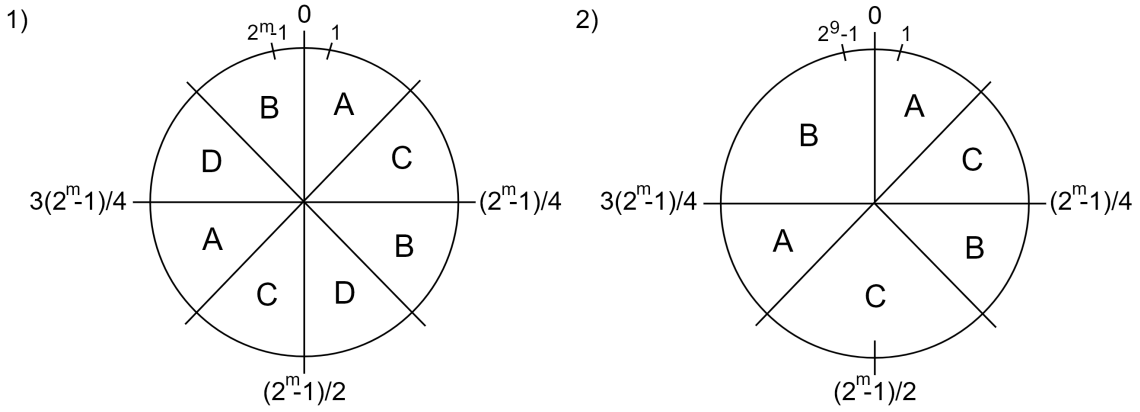


Figure 2.51: A consistent hashing ring of m-bit values, where we choose $k = 2$ splits for $n = 4$ servers across the ring. 1) All 4 servers and their replicated virtual nodes in even distribution. 2) Server D goes down, transferring its range to B and C.

In the above figure, choosing just $k = 2$ makes a huge difference as opposed to our previous solution in Figure (2.51).

2.10 MapReduce

It's 2004 and Google is looking for a way to process large amounts of web-crawled data efficiently in parallel. They found that most jobs follow the same pattern, leading to the following model:

Definition 10.1: MapReduce

MapReduce automatically parallelizes and executes client jobs provided they give these two functions:

- **Map**: Takes a set of input key-value pairs and produces a set of intermediate key-value pairs.
- **Reduce**: Takes an intermediate key and a set of values for that key, and merges them into a smaller set of values.

If you are familiar with functional programming, the MapReduce model is similar to the `map` and `reduce` functions as seen in languages like Python or JavaScript.

Example 10.1: Word Count in MapReduce

Input: A collection of documents (each document has a name and contents).
Output: The total frequency of each word across all documents.

Map:

- **Key:** document name
- **Value:** document contents
- **Emit:** for each word w in the document, emit $(w, 1)$, of form (key, value)

Reduce:

- **Key:** a word w
- **Value:** list of counts $\{1, 1, \ldots, 1\}$ from all maps
- **Emit:** $\left(w, \sum \text{counts}\right)$, i.e. the total occurrences of w

Example: {(A, "the dog likes to sit"), (B, "the lion does not sit")} maps to
{(the, 1), (dog, 1), (likes, 1), (to, 1), (sit, 1), (the,1) (lion, 1), (does, 1), (not, 1),(sit,1)}

This is then reduced to,
{(the, 2), (sit, 2), (dog, 1), (likes, 1), (to, 1), (lion, 1), (does, 1), (not, 1)}. ■

We can even stack multiple MapReduce jobs together. For example consider the below figure based on trying to find the set difference in Example (10.1):

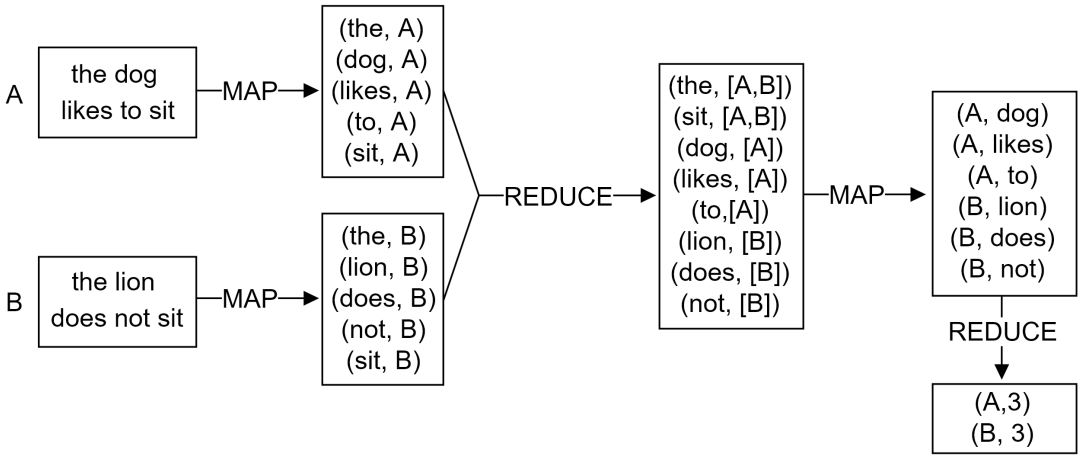


Figure 2.52: Both sets A and B under going two rounds of MapReduce. The first creates a set of words between A and B. The second round discards non-singletons, reducing the set to counts of elements in A and B.

Now to bring this into a distributed system:

Definition 10.2: Implementing MapReduce in a Distributed System

To implement MapReduce with a single coordinator as follows:

- **Split Data**: First take the input data and split it into M chunks.
- **Assign Maps**: The coordinator distributes the M chunks to N worker nodes (may receive multiple chunks).
- **Map Phase**: Each worker processes its chunk and partitions the output into R sections on disk. This is achieved by uniformly hashing the intermediate keys into R buckets.
- **Shuffle Phase**: The coordinator collects the R partitions from all workers and redistributes them back to N workers, via sorting the intermediate keys, or by hashing (more efficiently).
- **Reduce Phase**: Each worker processes its partition and writes the final output to disk.

Additionally, Given W workers, M mapping tasks, and R reducing tasks, $W \gg N$ and $R \gg N$. I.e., Mapping and Reducing tasks should outnumber the workers to keep them busy.

Now to deal with failures:

Definition 10.3: Fault Tolerance in MapReduce

We deal with hiccups in our system via the following:

- **Map/Shuffle Phase**: If a worker fails to map or goes offline with the intermediate data, the coordinator will reassign the chunk to another worker.
- **Reduce Phase**: If a worker fails to reduce, the coordinator will reassign the partition to another worker.
- **Coordinator Failure**: If the coordinator fails, the system will need to restart the entire MapReduce job. Failures are not recoverable, and are assumed to be rare.

If a worker is slow (**straggler**), the coordinator reassigns its task and reacts accordingly:

- **Map Phase**: The coordinator will only point to the first worker that finishes the map task for intermediate data.
- **Reduce Phase**: It doesn't matter who finishes first, as they write the same data to the same location on disk (e.g, "/filepath/final_data/id"). Moreover, writing is atomic.

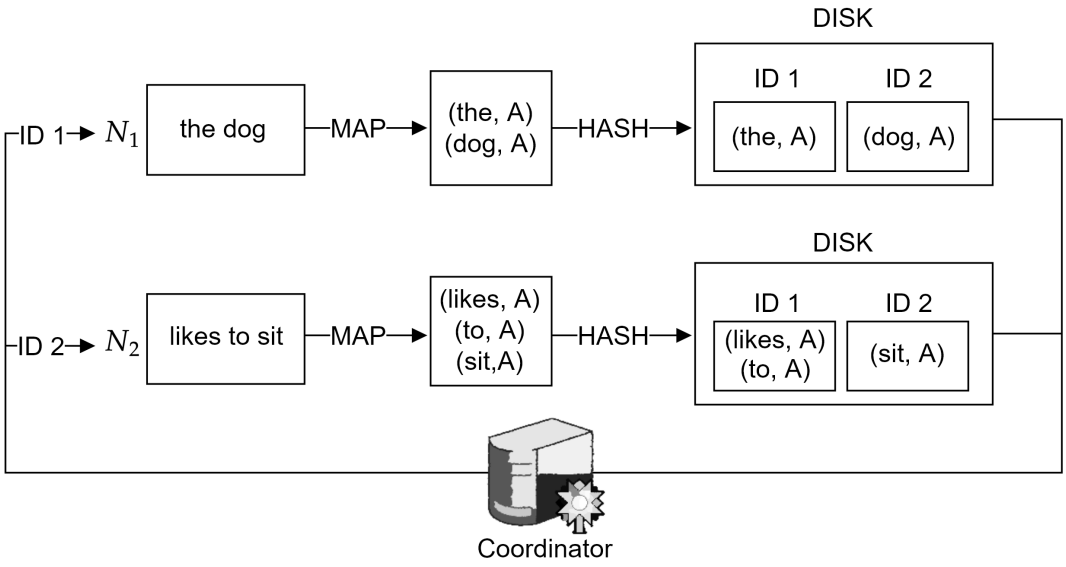


Figure 2.53: In a simplified MapReduce system, two workers receive a partitioned map job of "the dog likes to sit". After N_1 and N_2 finish processing the job, they hash the intermediate data into $R = 2$ buckets. The coordinator then collects the buckets assigning ID 1$\rightarrow N_1$ and ID 2$\rightarrow N_2$, to finish the reduce job.

2.11 Google File System (GFS)

Its 2003 and a research paper by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, lay out what would become the Google File System (GFS). In the early 2000s, Google rapidly expanded its workload, scaling to support services like, web search, indexing, and data analytics. Traditional file systems were not suited for their needs. Hence, GFS was designed for the following (Below you may find the GFS paper (Original Paper):

Definition 11.1: Design Goals of GFS

Google File System (GFS) is a distributed file system, which aims to have:

- **Massive scalability:** Store vast amount of data across thousands of inexpensive commodity servers.
- **High availability:** Tolerant to frequent component failures **(replication needed)**.
- **High throughput:** Optimize for large data-streams of concurrent sequential reads and a majority of append-only writes.

Consistency Model: Weak consistency model for improved performance.

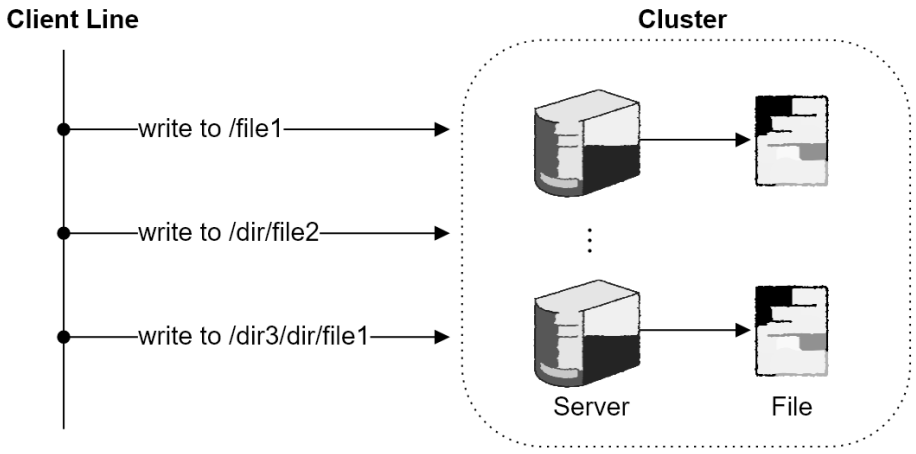


Figure 2.54: A high-level sketch of the Google File System (GFS) architecture.

Note: This is important as Google at the time was using cheap commodity hardware, which was prone to failure.

The following details how files are stored in GFS:

Definition 11.2: File Chunking

Files in GFS are divided into fixed-size chunks of 64MB. Each chunk is stored in a **chunk server**, identified by a unique 64 bit ID called a **chunk handle**.

To ensure fault tolerance, each chunk is **replicated** at least N times across different chunk servers (Typically $N = 3$).

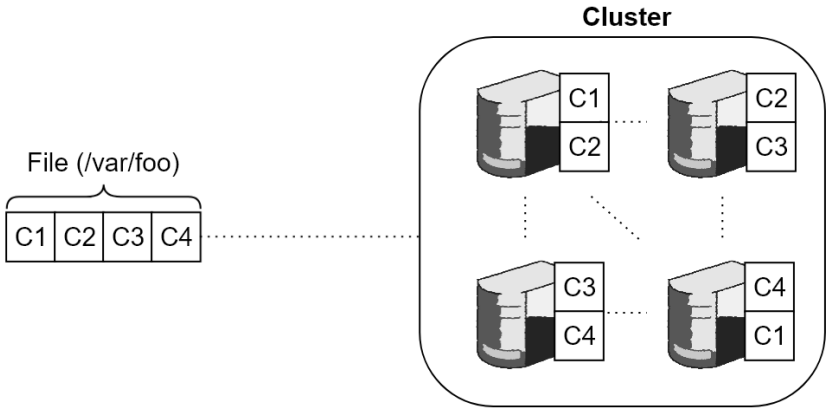


Figure 2.55: Simplified view of chunking in GFS, where a file is divided into 4 chunks (C1–C4), replicated twice across different servers.

We decide to store metadata on a dedicated server to act as our coordinator.

Definition 11.3: Metadata Management

The GFS **master** serves **one cluster**, storing the following metadata in memory:

- **File & Chunk Namespaces:** Hierarchical directory tree of files and the global chunk-ID namespace.
- **File→Chunk Mapping:** For each file, the ordered list of chunk handles.
- **Chunk Replica Locations:** Connecting chunk handles to their chunk servers holding its replicas via [IP:port].
- **Access Control Information:** File permissions and ownership attributes.

In particular, mappings are persisted via an operation log. Chunk locations are reconstructed by polling chunk servers at startup or when they join.

Now to discuss how client's read data from GFS:

Definition 11.4: Client Interaction

Clients interact with GFS via a two-step process:

- **Metadata Operations:** Clients query the master with the (file name, chunk index), receiving the chunk handle and its replica locations.
- **Data Operations:** The client caches such information and then directly communicate with the chunk servers.

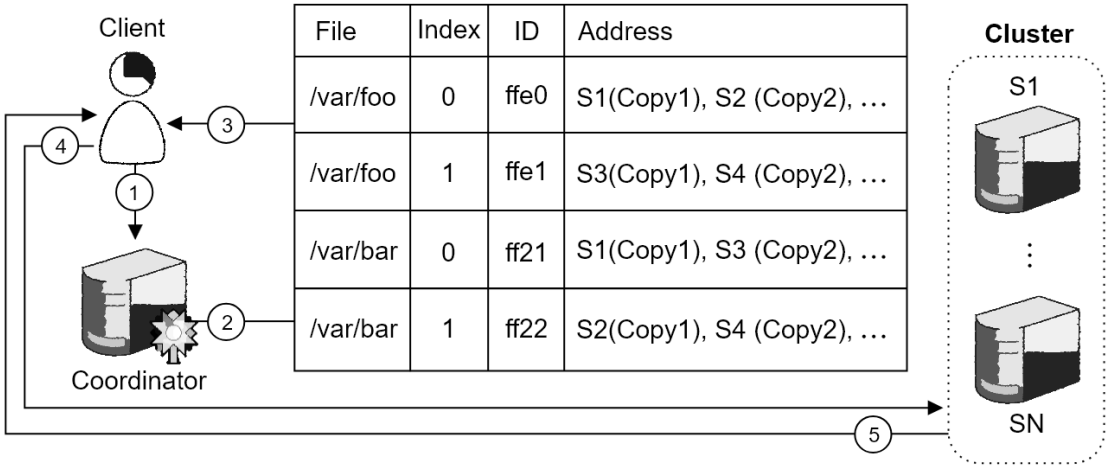


Figure 2.56: A Simplified view of client interaction with GFS. (1) Client sends a request to the master for metadata. (2) The master finds the chunk handle and its replica locations. (3) The master returns the chunk handle and replica locations to the client. (4) The client sends a request to the chunk server for data. (5) The chunk server returns the data to the client.

In our case we have specific needs that allow staleness to be tolerated:

Definition 11.5: GFS – Stale Reads are Allowed

Given the application of GFS's workload, it is acceptable to have stale reads. This greatly improves performance, as we don't have to have additional round trips to ensure the data is up to date.

For example, we may be gathering analytics which improve search results. Retrieving stale data doesn't break the system. When the latest data does arrive, we notice improved performance.

The client reads and writes data in two slightly different ways:

Definition 11.6: Read and Write Routines

Reads: The clients retrieves a server location from the master, reading directly from them.
Writes: Follow a three-step process:

- **Metadata Retrieval:** Clients query the master with the (file name, chunk index), receiving a list of replica locations. The master chooses one server to be the **primary replica**, granting it a **lease**, which defines the time period such server may act as the primary replica (typically 60s).
- **Data Transfer:** From the list of replicas, the client sends the data to the **closest replica** (not necessarily the primary replica). Such replica propagates the data to the next closest server, and so forth.
- **Commit Phase:** The client sends a **commit** request to the primary replica, which then notifies the other replicas to apply the changes. The primary replica then sends an **ACK** to the client.

In particular, the primary server follows a specific routine:

Definition 11.7: Primary Server Routine

The **primary** replica orchestrates concurrent writes with strict per-mutation ordering:

- **Lease Validation:** Upon each `WriteChunk` RPC, check that the primary's lease (granted by the master) is still valid. If it has expired, reject the request so the client can **refresh** metadata.
- **Replica-log ACKs:** The client waits for all replicas to ACK that the log has been replicated. Then it tells the primary to commit.
- **Per-Mutation Sequencing & Serialization:** Each incoming write is assigned a consecutive sequence number (possibly from multiple clients), which will be applied in sequential order.
- **Commit and Reply:** On command, the primary server applies the mutation in serial order, telling the secondaries to do the same; **However**, the primary sends the ACK to the client immediately even before any replicas even acknowledge the commit.

  This means **stale data** may be latent in the system. Though, our master will eventually catch this, and re-replicate the data.
- **Error Handling:** If any secondary fails to replicate the log in time (not the apply phase), abort the mutation, notify the client, and rely on the client's retry logic.

We illustrate the write process in the figure below:

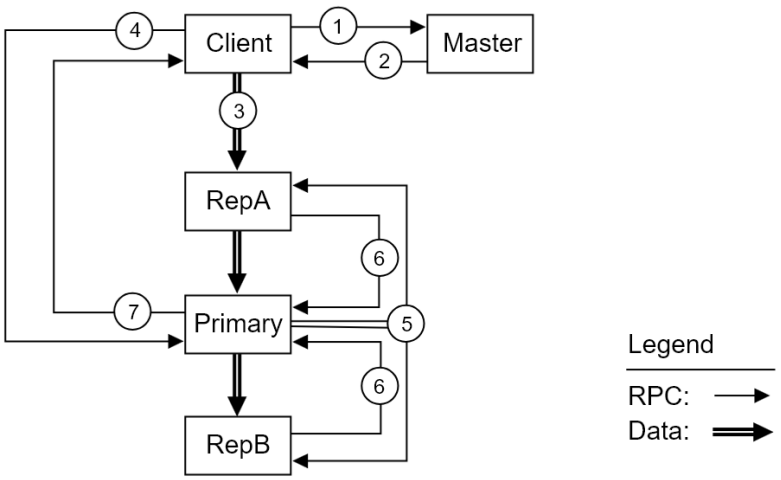


Figure 2.57: A Simplified view of the write process in GFS. (1) Client sends a request to the master for metadata. (2) The master finds the chunk handle and its replica locations. (3) The client sends a request to the closest replica who propagates it through the network. (4) After receiving ACKs from all replicas, the client sends a commit request to the primary replica. (5) The primary replica sends a commit request to all replicas. (6) The replicas send ACKs back to the primary replica. (7) The primary replica sends an ACK back to the client.

Now we need to ensure there are N copies of a chunk at all times:

Definition 11.8: Chunkserver Heartbeat Routine

Chunkservers send periodic **HeartBeat** RPCs to the master, conveying:

- **Held Chunk Handles & Versions:** The set of chunk IDs stored locally along with each chunk's current version, so the master can detect **missing or stale replicas**.
- **Lease Extension Requests:** Any primary replicas include lease-renewal requests for the chunks they lead.

The master's response piggy-backed on the heartbeat (reply to such heartbeat) may grant new leases or issue commands (e.g. initiate re-replication). If a chunk server's heartbeat is not received within a timeout, the master marks it dead and schedules re-replication to restore N live replicas per chunk.

The master handles persistent state and snapshots for log reduction in the following way:

Definition 11.9: Master Checkpoint & Operation-Log Routine

The **master** persists its metadata through two complementary mechanisms:

- **Operation Log:** An append-only record of every metadata-mutating request (e.g. file/directory creation, chunk allocation). Each entry is synchronously written to the master's local disk and replicated to remote machines.
- **Periodic Checkpoints:** In a background thread, the master periodically snapshots its entire in-memory state (file and chunk namespaces plus file→chunk mappings), writes the checkpoint to disk, and safely truncates the operation log up to that point.

On startup (or after a crash), the master loads the latest checkpoint and then replays any subsequent operations from the log to reconstruct its full metadata state.

In short, the master must keep record of any critical changes to the metadata, and sufficiently back it up to ensure it can recover. Though, having a single master is a bottleneck. Hence, we keep a replica in the background:

Definition 11.10: Shadow Master

A **shadow master** is a standby replica of the GFS master that provides high availability:

- **Operation-Log Mirroring:** Maintains a copy of the master's operation log, replicated remotely.
- **State Replay:** Continuously **replays** (applies) logged operations to keep its in-memory metadata (namespaces and file→chunk mappings) nearly up to date.
- **Read-Only Serving:** Can answer client metadata queries (e.g. lookups) in a read-only fashion, offloading the active master.
- **Failover Ready:** On active-master failure, a simple DNS switch (switching the network pointer to the master) promotes the shadow to become the new active master.
- **Eventual Consistency:** May lag slightly behind the active master due to asynchronous log replication and replay.

Though we must point out:

Theorem 11.1: Non-Atomicity and Non-Serializability Across Chunk Boundaries

In GFS, writes that span multiple chunks are neither atomic nor globally serializable. Concretely, there exist two concurrent write operations W_1 and W_2 and chunk indices $i \neq j$, such that the primary for chunk i orders W_1 before W_2, while the primary for chunk j orders W_2 before W_1.

However, concurrent writes are atomic and serializable **within a single chunk**.

To point out, and emphasize:

Definition 11.11: One Primary for each Chunk

In GFS, each chunk is assigned exactly one **primary** replica by the master via a lease.

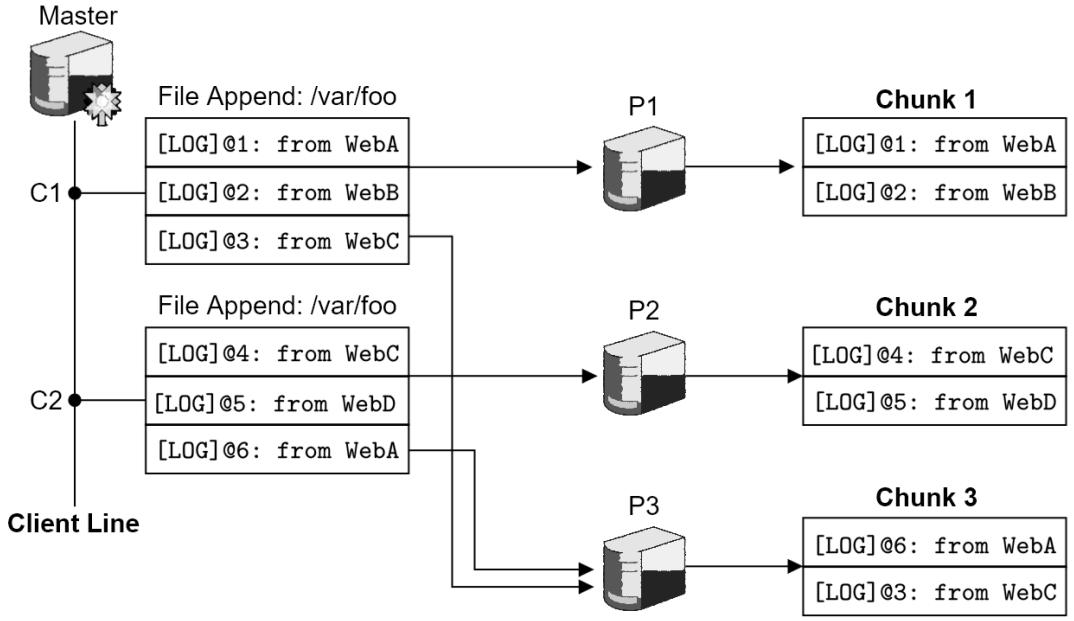


Figure 2.58: GFS record-append across chunk boundaries (client→master→client→primary). Each chunk holds up to two log records. Clients C_1, C_2 issue appends `@1`–`@6`. Under primary P_1 (chunk 1), `@1`,`@2` succeed while the rest are rejected. Each client then re-queries the master and retries on primary P_2 (chunk 2), where `@4`,`@5` succeed but `@3`,`@6` are rejected. A final refresh to primary P_3 (chunk 3) accepts `@6`,`@3` in arrival order, illustrating that independent primaries can impose different orders, and that multi-chunk appends are non-atomic and non-serializable.

Below we summarize the key points of GFS through an illustration:

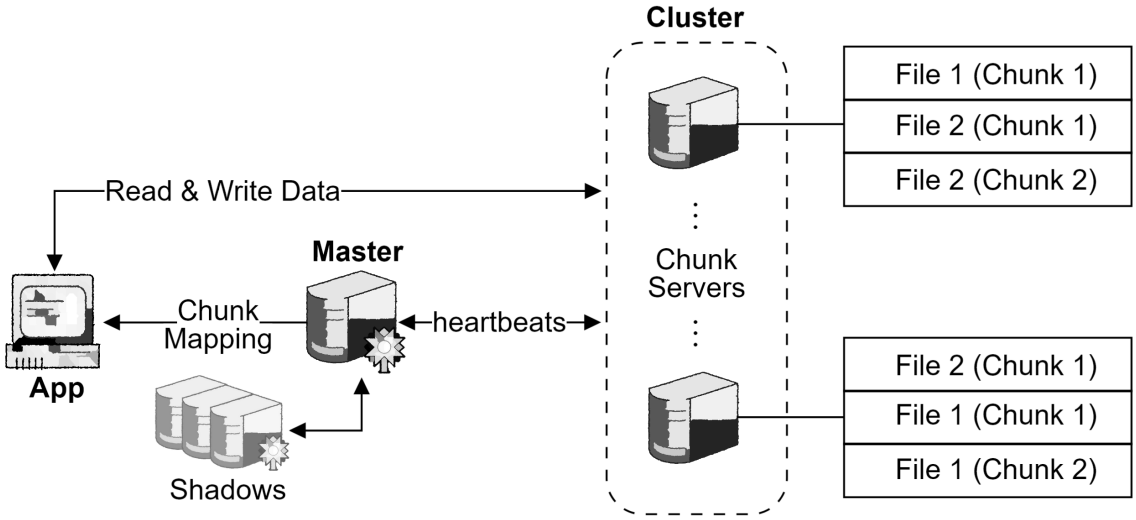


Figure 2.59: High-level GFS architecture. An application (`App`) first contacts the **master** to obtain file→chunk mapping (via file name and chunk index), then directly reads from or writes to the appropriate **chunk servers**, which store fixed-size, replicated chunks (e.g. chunks of File 1 and File 2) across the cluster. The master keeps all metadata in memory, persists mutating operations in an append-only operation log, and receives periodic **HeartBeat** RPCs from chunk servers to monitor replica state and grant leases. **Shadow masters** asynchronously replay the operation log to maintain a nearly up-to-date, read-only metadata copy for load-sharing and fast failover.

2.12 Dynamo: Amazon's Highly Available Key-Value Store

It's 2007 and Amazon's global e-commerce platform must remain "always-on" despite continual component failures, outages cost revenue and customer trust. To achieve this, Dynamo sacrifices strict consistency for low-latency availability (Original Paper):

Definition 12.1: Design Goals of Dynamo

Dynamo is a decentralized, highly available key-value store designed to:

- **Always-Writeable:** Never reject writes under partitions or node failures, deferring conflict resolution to the read path.
- **Incremental Scalability:** Scale out by adding or removing nodes without downtime or manual repartitioning.
- **Decentralized Symmetry:** No central coordinator, based on Consistent Hashing (2.9).
- **Low-Latency Performance:** Dynamo provides its clients an SLA (Service Level Agreement) that under any load, it provides a 300ms response 99.9% of the time.
- **Eventual Consistency:** Allow temporary inconsistencies under failures, ensuring all updates reach replicas eventually.

Consistency Model: Weak Consistency, favoring availability over strict consistency.

Definition 12.2: Quorum System – Coordinating (Part 1)

A user's keys is hashed to a point on a 128-bit ring and mapped to an ordered list of N nodes (the **preference list**). The first clockwise node, N_i, becomes the **coordinator**:

- **Writes**: Coordinator N_i receives and sends `put(key,value)` in parallel to the next top $N-1$ replicas, waits for acknowledgments from any W distinct replicas (**including itself**), then returns success to the client.
- **Reads**: N_i receives and sends `get(key)` to all $N-1$, returning success after R nodes acknowledge the read.
- **Conflicts**: Divergent data is reconciled via a vector-clock versioning history. If the vector clocks cannot be merged, it is left to the **client** to resolve.
- **Quorum Condition**: Choosing parameters $R + W > N$, ensures R and W overlap, diminishing staleness.
- **Ownership:** If key k hashes to A's segment, A is the **primary** coordinator when it's healthy. If A is unreachable, the **next alive** node temporarily stands in as coordinator.

We'll call the below a turtle-back diagram; it illustrates the quorum system in a basic configuration:

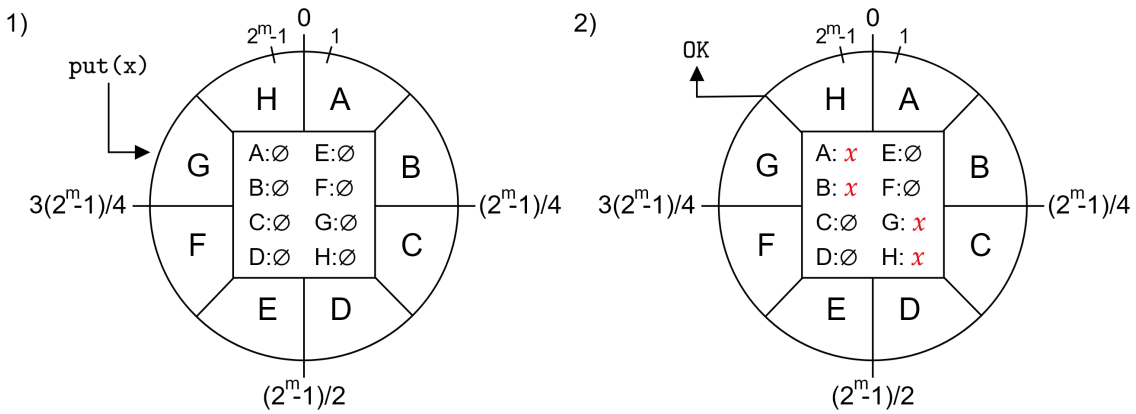


Figure 2.60: A basic quorum system. **To be clear**: virtual nodes are positioned at the spokes. For instance, H starts at 0, and ends at the next left-spoke G (the top-left corner of the center box). Here, $R = 4, W = 4, N = 7$, and servers are an ordered list of virtual nodes A–H. Also, here we say `put(x)` for brevity, while it's actually `put(key,value)`. 1) A client's put request falls within G's range. 2) The next top $W - 1$ nodes acknowledge, with G sending back an `OK` (success).

Moving on, we deal with the liveness of nodes, and how to reconcile data:

Definition 12.3: Quorum System – Gossip & Hints (Part 2)

Dynamo's monitors liveness with the following mechanisms:

Gossip Frequency & Peer Selection: Every second, each node picks a random peer and exchanges its local membership-change log (join/leave/failure records). This gossip ensures that all nodes eventually learn who is up or down without any central service.

Failure Detection & Sloppy Quorum: During a write, if a coordinator cannot reach a preferred replica (due to a failure or partition), it writes to the next healthy node in the preference list. That node stores the update alongside a **"hint"** tagging the intended replica.

In particular, each node stores hints in a separate local database unbeknownst to the client. This is called a **sloppy quorum**: it allows writes and reads to proceed even if some replicas are unreachable, as long as W and R are satisfied.

Hinted Handoff: When the failed node rejoins, any node holding a hint for it will detect its liveness via gossip and then forward the update it missed in a "handoff"—restoring the full set of N replicas **without** blocking client operations.

Below illustrates the gossip protocol and hinted handoff:

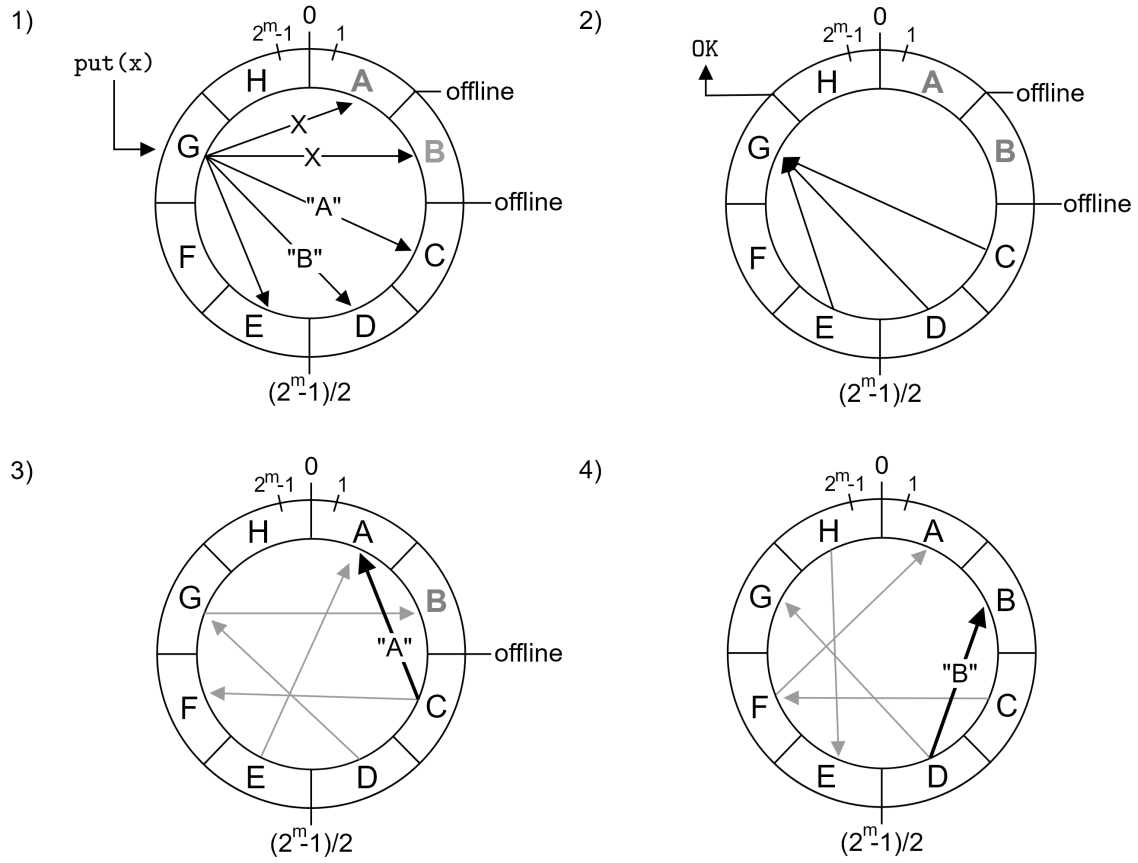


Figure 2.61: Gossip protocol and hinted handoff. 1) A put operation is sent to G, from which is then propagated. Though, A and B appear to be down, G resolves this by giving hints to C and D. 2) Nodes C, D and E `ACK`, allowing G to return `OK`. 3) During the gossip protocol A comes back online; C notices this and sends the hint to A (hinted-handoff). 4) Later in the gossip protocol, B comes back online; D notices and hands off the hint to B. **Note**: The paper does not follow the one-hint-per-replica assumption made in this figure. Also, G **should actually** send the write in parallel to $H \to A \to B \to \ldots$ (its first three successors), but for visual-clarity we began with A.

Note: The Dynamo paper doesn't bound hint capacity. Here we assume at most one hint per missing replica, which in the worst case could lose writes if that holder also fails. We might instead store hints on the next N healthy nodes, or allow multiple distinct hints per replica to improve safety.

The paper does say, "replicas will keep [hints] in a separate local database that is scanned periodically." The use of the word **Database** suggests hints may be propagated generously.

Now we discuss in detail how dynamo utilizes **Vector Clocks** to reconcile divergent data:

Definition 12.4: Quorum System – Vector Clocks & Reconciliation (Part 3)

Every object (key-value pair) in Dynamo is associated with a **vector clock** VC that tracks the version history of the object. The VC is composed of N (node, counter) tuple pairs, maintaining a change history:

- **Stamping & Propagation:** A client invokes $\texttt{put}(k, v, VC_{\text{ctx}})$ at the coordinator N_i, where VC_{ctx} is the VC from the last read (`get()`). N_i merges VC_{ctx}, increments its own counter entry in VC, stores (k, v, VC) locally, and then sends the entire tuple (k, v, VC) in parallel to the other $N - 1$ replicas.
- **Primary Path:** In the failure-free case, N_i waits for W acknowledgments (including itself) before replying success. Each replica simply stores the tuple under key k in its local key-value store.
- **Failover Path:** If N_i is unreachable, the client retries $\texttt{put}(k, v, VC_{\text{ctx}})$ at the next alive node in the preference list. That node becomes the temporary coordinator: it fetches or merges existing clocks, stamps its own counter, performs a sloppy-quorum write of (k, v, VC), and tags hints for any skipped replicas (including the primary).
- **Read Reconciliation:** A client issues $\texttt{get}(k)$ to the coordinator, which polls all N replicas and waits for the first R responses (v_j, VC_j). If one VC_j strictly dominates the others, its v_j is returned; if some VC_j are concurrent, all corresponding v_j are returned.
- **Application Merge:** When multiple versions (v_j, VC_j) arrive, it's up to the application to merge them (semantic reconciliation) into a single v_{merged} and VC_{merged}, then issue a fresh $\texttt{put}(k, v_{\text{merged}}, VC_{\text{merged}})$.

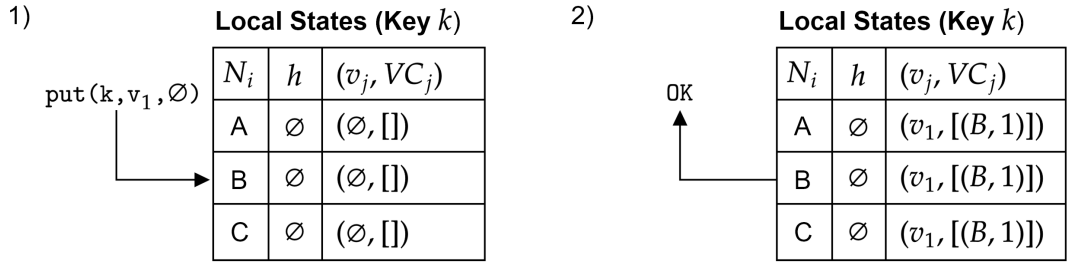
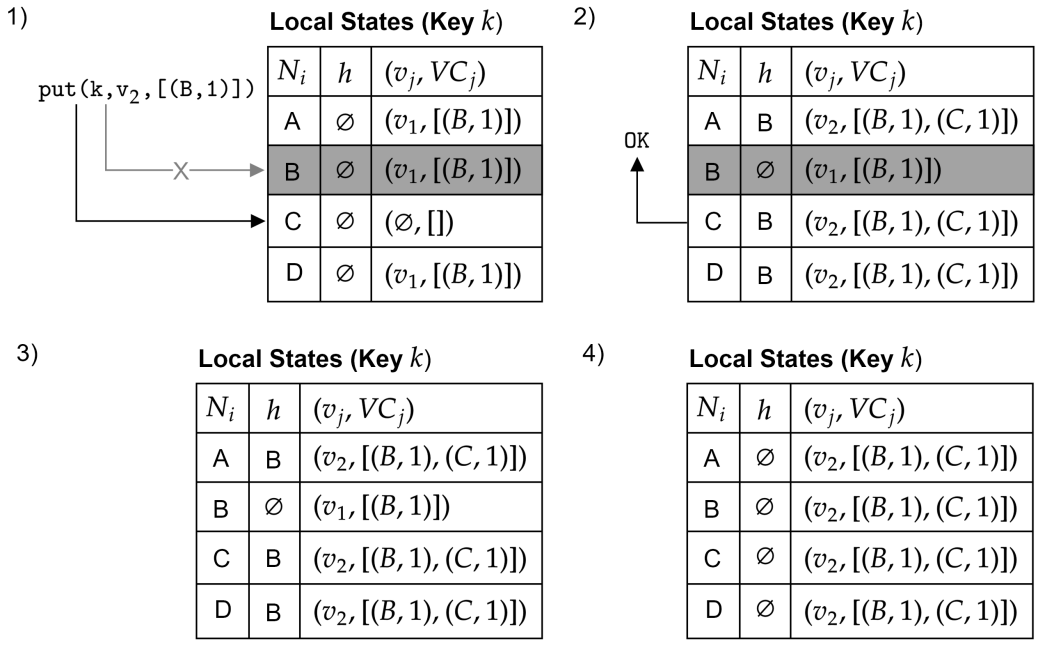


Figure 2.62: A simplified view of a ring with three N_i nodes, A, B, and C, h hints, and a (value, vector clock) tuple (v_j, VC_j). 1) A client issues their first put request of $\texttt{put(k,v}_1\texttt{,}\varnothing\texttt{)}$ (empty context) to B. 2) B stores the tuple, $(v_1, [(B, 1)])$, indicating the new value, and notes its participation in a vector clock VC. It propagates (k, v_1, VC) to A and C, returning success to the client.



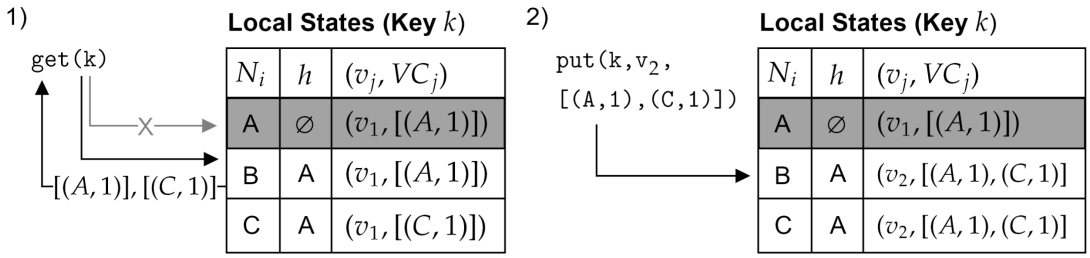
N_i	h	(v_j, VC_j)
A	∅	$(v_1, [(B,1)])$
B	∅	$(v_1, [(B,1)])$
C	∅	$(\emptyset, [])$
D	∅	$(v_1, [(B,1)])$

N_i	h	(v_j, VC_j)
A	B	$(v_2, [(B,1),(C,1)])$
B	∅	$(v_1, [(B,1)])$
C	B	$(v_2, [(B,1),(C,1)])$
D	B	$(v_2, [(B,1),(C,1)])$

N_i	h	(v_j, VC_j)
A	B	$(v_2, [(B,1),(C,1)])$
B	∅	$(v_1, [(B,1)])$
C	B	$(v_2, [(B,1),(C,1)])$
D	B	$(v_2, [(B,1),(C,1)])$

N_i	h	(v_j, VC_j)
A	∅	$(v_2, [(B,1),(C,1)])$
B	∅	$(v_2, [(B,1),(C,1)])$
C	∅	$(v_2, [(B,1),(C,1)])$
D	∅	$(v_2, [(B,1),(C,1)])$

Figure 2.63: 1) A client issues a put request with `(k,v2,[(B,1)])` implying a previous interaction of to B. Though, B is down, and C is the next alive node; However, C has never seen k before. C takes the user's dominating input. 2) C stores the tuple, $(v_2, [(C,1),(B,1)])$, indicating the new value, and notes its participation in a vector clock VC. It propagates (k, v_2, VC) with hint B to D and A, returning a success. 3) Later, B rejoins during gossip. 4) A participants notice B's revival, sending it updates.



N_i	h	(v_j, VC_j)
A	∅	$(v_1, [(A,1)])$
B	A	$(v_1, [(A,1)])$
C	A	$(v_1, [(C,1)])$

N_i	h	(v_j, VC_j)
A	∅	$(v_1, [(A,1)])$
B	A	$(v_2, [(A,1),(C,1)]$
C	A	$(v_2, [(A,1),(C,1)]$

Figure 2.64: 1) The client requests key k from A, but it's down, falling over to B, which gets a read from C; However, C's entry is divergent, perhaps due to a network partition. To resolve, B sends back both vectors, relying on the client to reconcile the data. 2) The client mends the data by merging the two vectors, pushing it back to B. For example, a customer may have had two tabs open with different shopping carts. The application decides to merge the two carts, adding to the user experience.

In an eventually-consistent store, vector clocks track the history of individual object updates but cannot tell us which keys out of millions have diverged across replicas. Dynamo employs the following technique to solve this problem efficiently:

Definition 12.5: Anti-Entropy via Merkle Trees

Dynamo uses Merkle trees to detect and repair divergent replicas without exhaustive scans:

- **Tree Construction:** Each node builds a binary Merkle tree over its local key-value store. **Leaves** are the cryptographic hashes of individual key-value pairs (or small fixed-size batches of adjacent keys), computed as

  $$H_i = \text{hash}(k_i \| v_i)$$

  Then each **internal node** hashes the concatenation of its two children, where ($\|$) denotes concatenation, and H is a cryptographic hash function (e.g., SHA-1, MD5). The root hash summarizes the entire tree.

- **Root Comparison:** Replicas exchange only their root hashes. A match means the entire key-value set is identical—no further work.

- **Recursive Descent:** On a root mismatch, only the child hashes of the differing subtrees are exchanged, descending the tree until the exact leaf hashes (and thus specific keys) that differ are isolated.

- **Selective Synchronization:** Once out-of-sync keys are pinpointed, only those individual (k, v) pairs (and their vector-clock metadata) are transferred and reconciled.

- **Efficiency Gains:** By limiting data exchange to mismatched branches, this method drastically reduces both network bandwidth and disk I/O compared to full scans.

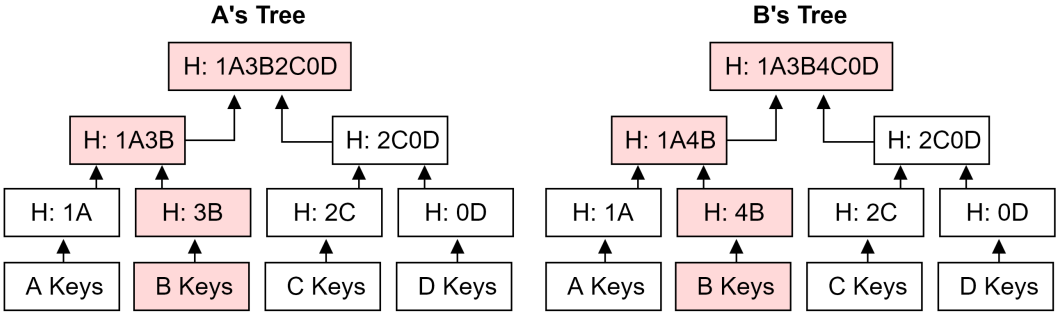


Figure 2.65: Given some arbitrary simplified hashing function, we construct a Merkle tree over key ranges A–D. Here A and B compare root hashes. They differ, so they descend to the branch which causes the mismatch. They both discover a discrepancy in their B key ranges.

2.13 Spanner: A Globally-Distributed Database

In 2012, Corbett et al. introduced Spanner, Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. The first system to support world-wide atomic transactions with its novel TrueTime API (Original Paper):

Definition 13.1: Spanner Design Goals

Google's Spanner is architected to be a planet-scale, strongly-consistent system. Its primary design goals are:

- **Global Scale & Distribution:** Span millions of machines and petabytes of data across multiple datacenters, automatically sharding and rebalancing data.
- **External Consistency:** Provide linearizable reads and writes across shards and datacenters via TrueTime; every transaction appears to occur at a single, globally-agreed timestamp.
- **Serializability:** Support serializable multi-row, multi-shard transactions using two-phase locking within Paxos-replicated groups (a consensus model comparable to raft) and two-phase commit across groups.
- **High Availability & Durability:** Synchronously replicate each shard with Paxos so that any replica group tolerates datacenter outages without losing committed data.
- **Low-Latency, Lock-Free Reads:** Enable read-only transactions to serve at a timestamp in the past without acquiring locks, minimizing read latency under contention.

In short,
Consistency Model: Strict serializability, with atomic transactions.

Note: We do not cover paxos, but every time it is said, just think of it as the consensus module in the system. I.e., one may synonymously think of it as raft.

The main motivation here is that when we want to do a database backup, or perform some enourmous read, placing a lock on the entire system would not be a great user experience. So we utilize snapshots instead, recall Definition (1.2):

Let's discuss how Spanner achieves these goals, with snapshots and timestamps:

Definition 13.2: Consistent Snapshots via MVCC

Spanner uses **Multi-Version Concurrency Control (MVCC)** and globally ordered timestamps to serve lock-free, causally-consistent read-only transactions:

- **Versioned Writes:** Each update transaction T_w is assigned a unique, monotonically increasing commit timestamp t_w. Every write to key k creates a new version tagged with t_w rather than overwriting.
- **Safe Time Rule:** Before T_r reads key k, it must see a write T_w that is $t_w > t_r$.
- **Causal Consistency:** Global timestamp ordering ensures that if $T_1 \to T_2$, any snapshot that reflects T_2's writes also reflects T_1's.
- **Lock-Free Reads:** Read-only transactions never acquire locks and run entirely against their fixed snapshot, eliminating read-write contention.

Definition 13.3: Commit Timestamp Ordering

Spanner assigns each transaction T_i a commit timestamp t_n s.t., for any two T_1 and T_2:

$$T_1 \text{ completes before } T_2 \quad \Longrightarrow \quad t_1 < t_2.$$

We use **physical clocks** (atomic) to assign timestamps. Lamport (logical) clocks capture causal dependencies but **do not guarantee** real-time precedence.

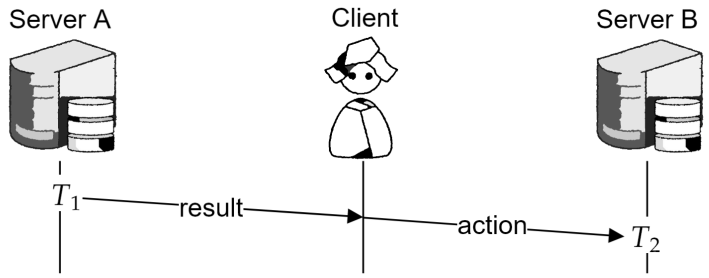


Figure 2.66: Server A may never communicate with server B, making logical clocks insufficient.

Google implements atomic clocks and GPS receivers to mitigate physical limitations of clocks:

Definition 13.4: TrueTime: Bounded Clock Uncertainty

Spanner's TrueTime API provides each server a timestamp interval $[t_{earliest}, t_{latest}]$ such that the actual absolute time (real-time) t_{abs} lies within the interval: $t_{earliest} \leq t_{abs} \leq t_{latest}$. On commit, Spanner enforces strict real-time ordering by:

- **Interval Return:** A call to `TT.now()` returns $[t_{earliest}, t_{latest}]$.
- **Uncertainty Wait:** Before finalizing a commit timestamp t, the coordinator waits until its local clock $\geq t_{latest}$, ensuring no future transaction can obtain a timestamp $\leq t$.
- **Monotonicity & Ordering:** This wait ensures that once a transaction reports commit at t, all subsequent `TT.now()` calls on any server will yield intervals with $t_{earliest} > t$, guaranteeing T_1 commit precedes T_2 timestamp.
- **Global Coverage:** By combining redundant GPS receivers and local atomic clocks in each datacenter, TrueTime keeps its uncertainty bound $\delta = t_{latest} - t_{earliest}$ small (typically a few milliseconds), even under network partitions (Kops)

E.g., If it's 12:32:01 pm we might see: [12:31:59pm, 12:32:04pm] where δ is 5 milliseconds.

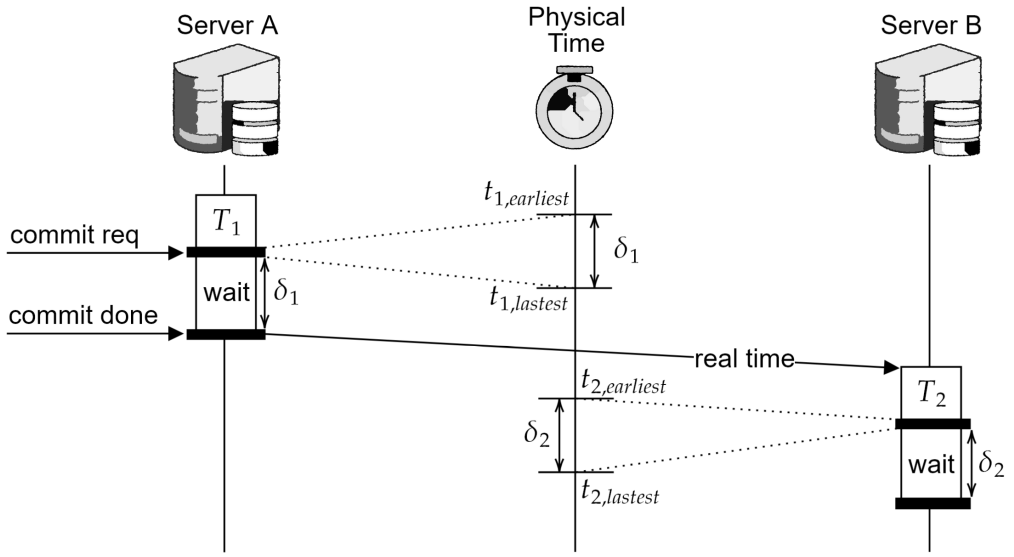


Figure 2.67: Upon receiving a commit request, each coordinator (e.g. A) calls `TT.now()` and obtains an uncertainty interval $[t_{1,\text{earliest}}, t_{1,\text{latest}}]$. It then delays for $\delta_1 = t_{1,\text{latest}} - t_{1,\text{earliest}}$ before finalizing its commit timestamp.

Finally we touch on the two-phase commit protocol:

Definition 13.5: Cross-Shard Transactions & Paxos Replication

Spanner provides strictly serializable, atomic transactions across independently-replicated shards by combining two-phase locking (2PL), two-phase commit (2PC), and Paxos state-machine replication:

- **Coordinator Selection:** After a client buffers all its reads and pending writes, it picks one of the involved shards' Paxos group leaders to act as the **2PC coordinator**. This is simply the leader replica for one of the keys in the transaction's shard set.
- **Intra-Shard Isolation:** Within each shard, that shard's Paxos leader acquires two-phase-locks on the keys being accessed, serializing concurrent read-write operations.
- **Atomic Cross-Shard Commit:** The coordinator issues a 2PC "`PREPARE`" to every participant leader. Each participant logs the prepare record via Paxos, acquires its write-locks, and returns a prepare-OK. Once the coordinator has all prepares, it logs the global "`COMMIT`" via Paxos to each shard, ensuring the transaction is made durable and atomic.
- **Paxos Fault Tolerance:** Let f be the maximum number of servers allowed to fail before system failure. Then each shard is a Paxos group of $n = 2f + 1$ replicas; as long as a majority $(f + 1)$ are up, Paxos ensures progress, automatically electing new leaders and replicating log entries even if up to f replicas fail.

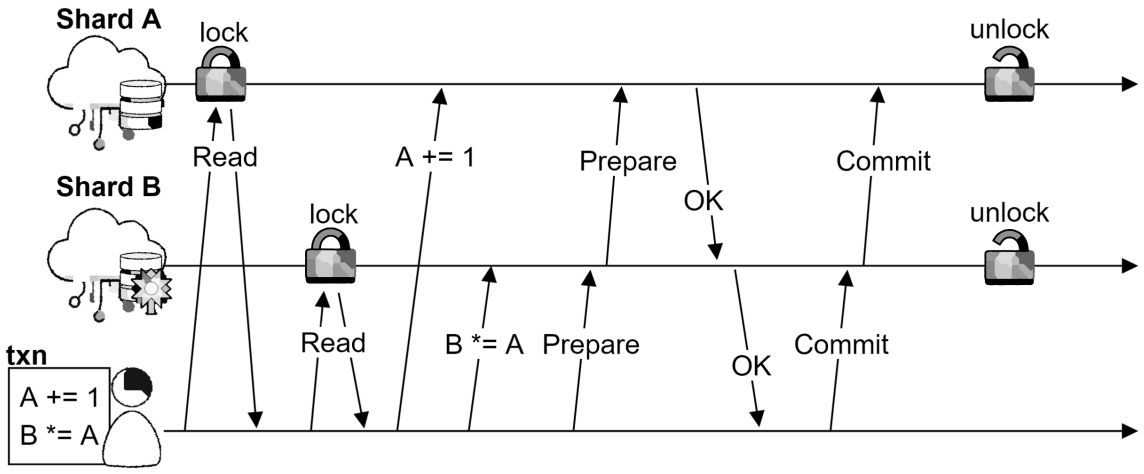
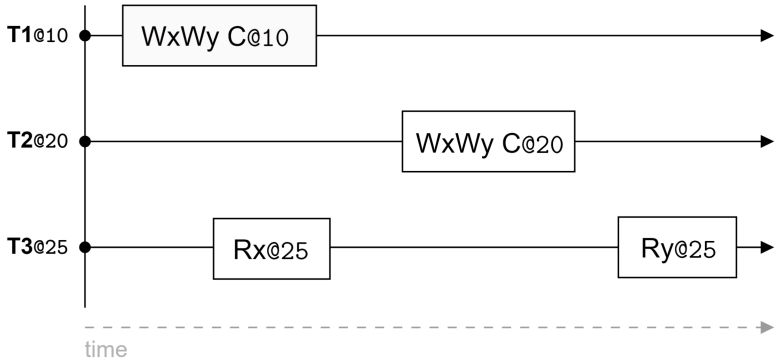


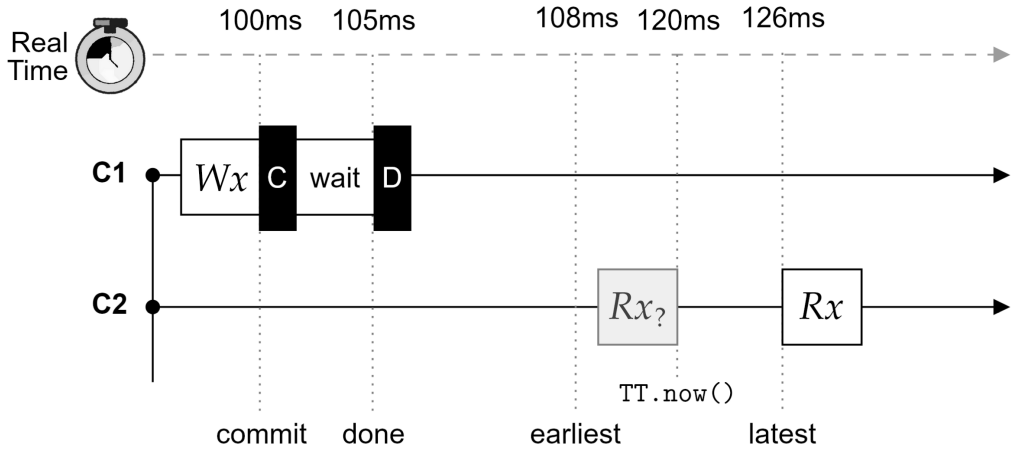
Figure 2.68: A Read-Write Transaction 2PC Commit in Spanner, where txn is the transaction, B's Paxos leader is the coordinator, and A another participant. First, reads acquire locks, then 2PC begins. Commands are sent, and then the `PREPARE` and `COMMIT` phases begin, ending with the release of locks.

Example 13.1: MVCC & TrueTime Evaluation

Consider the bellow transactions:



Here we have three transactions, T_1, T_2, and T_3. First we see that T_1 make a write request to both keys x and y at timestamp 10. Then T_2 makes the same request but at timestamp 20. Though it appears T_3 is a bit out of sync, reading at time 25. Hence, by the safe time rule, T_3 must see a write propagate with a time greater than 25. Now adding TrueTime intervals:



Here client C_1 has already prepared the write Wx. C_1 then sends a commit request at 100 ms, which returns a TrueTime interval of [100ms, 105ms]. The black boxes visualize possible uncertainty intervals. C_1 then waits for 5 ms before sending the commit request. On line 2, C_2 sends a read request to x at 120 ms. They receive a TrueTime interval of [108ms, 126ms]. C_2 then waits for 6 ms before sending before attempting to read x; However, since C_1's commit timestamp is 105ms, C_2 must wait for a later commit timestamp, which is typically an insignificant amount of time. ■

2.14 TLA + & Closing Remarks

TLA + is a high-level specification language based on temporal logic that lets you model distributed algorithms, exhaustively explore all possible states, and prove safety and liveness properties. While it excels at uncovering subtle bugs and proving correctness, TLA + specifications do not translate directly into production code—implementing a verified design remains a substantial engineering effort.

Recent work at Boston University on **choreography coding** aims to close this gap: engineers write global protocol descriptions in a TLA +-like notation, and an automated toolchain generates the per-participant code that implements the protocol.

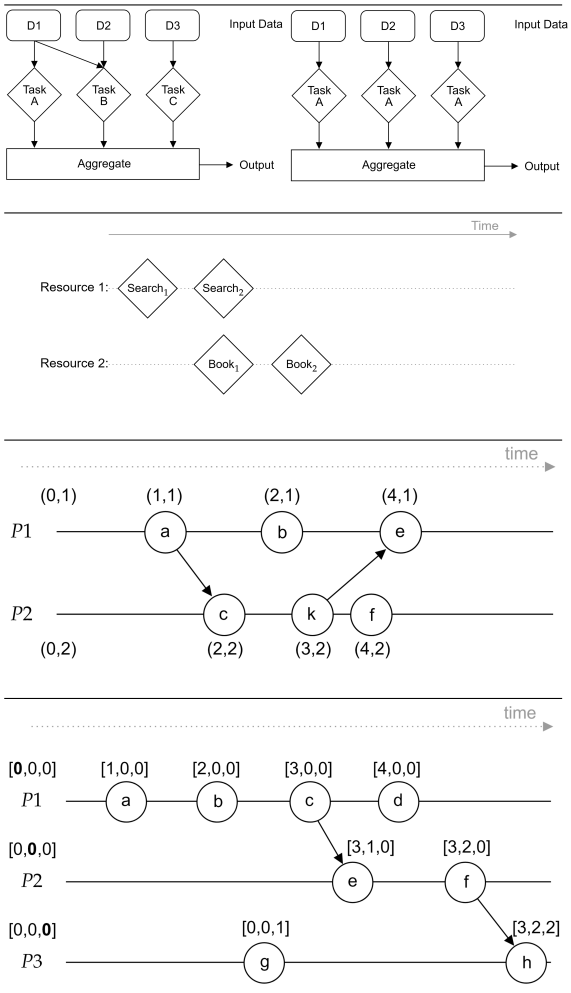
Final Note: To the other students reading this, I hope you find these notes helpful, and no we don't need to know how to code in TLA + for the exam. Just that it's not magic (can't generate code), but it can prove correctness.

Summary

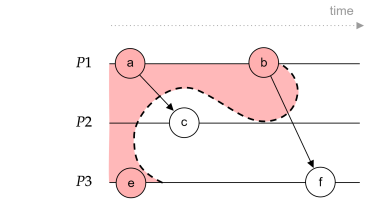
This page is left intentionally blank.

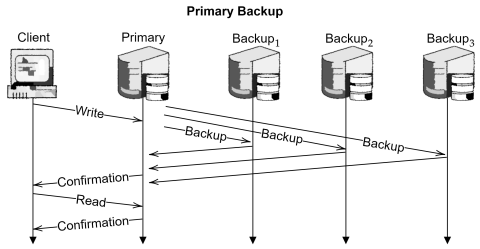
—**System** (1.1)—**Threads:** Hardware threads (CPU cores) each may run one software thread (program) at a time. Switching, between threads is context switching (overhead). E.g., Go manages internal thread pools, offering it to the OS, reducing overhead. **Thread Comm:** Interprocess communication (IPC), threads communicate via shared memory (e.g., channels, pipes, virtual memory). **Conc. & Parrelism:** Concurrency shuffles tasks, parallelism runs tasks simultaneously (threads). **Dist. Conn:** A client connects to a server (client-server model), a TCP conn. (FIFO) secures the line, the RPC (remote procedure call) abstracts dist. communication. **Races & Deadlocks:** Data race, two threads manipulating shared data (reads-onlys are fine). **Mutexes:** Placing locks around shared data, stopping concurrent access. **RPC Fail Models:** At-least-once, client retries until a response is received. Rads are fine, writes cause race conditions. At-most-once, server handles dupes, clients send unique IDs (cached responses). Exactly-once, both at-least-once and at-most-once. **(a)sync:** Asynchronous, non-blocking, no waiting for a response. Synchronous, blocking, waiting for a response. **(un)buffered-channners:** Unbuffered, sender waits for receiver on some thread to receive message. Buffered, sender sends message(s) to a buffer, takes one at a time. **Task, Data, Pipeline Parallelism:** Task, same data, different tasks. Data, same task, different data. Pipeline, task split into dependent stages, independent stages run in parallel. **Time Accr:** Physical clocks drift due to hardware limitations. Atomic clocks, have insignificant drift. NTP (network time protocol), utilizes GPS satellites to sync time, to a ground truth clocks, which propagates time to other systems.—**Logical Time** (1.3.2)—**Lamport Clocks:** Lamport Clocks, (t_p, p), t_p time of process p, monotonically increases for each event/send (Total Ordering). Receivers q resolve time differences, $t_q = max((t_p + 1), t_q)$ (send vs. local time). Given two events a & b, timestamps $t(a)$ & $t(b)$, with r trace: $a \rightarrow_r b \implies t(a) < t(b)$ (causal ordering); $t(a) \geq t(b) \implies a \not\rightarrow_r b$ (concurrent); $(t(a) = t(b) \wedge a \gg b) \implies a \rightarrow_r b$, s.t., $(\gg)$ rep. process order. **Non-causal:** $(a \ll b) := (a \not\rightarrow_r b) \wedge (b \not\rightarrow_r a)$ (concurrent). **Vector Clocks:** Operate as an array of Lamport clocks, index is process p_i, and value is t_{p_i} (time); However, sends do not increment t_{p_i}. Given timestamps a & b, if all indexes in a are larger than b, $a \rightarrow_r b$. If some in a are larger than b, vice versa, $a <> b$ (non-comparable, concurrent, partial ordering).

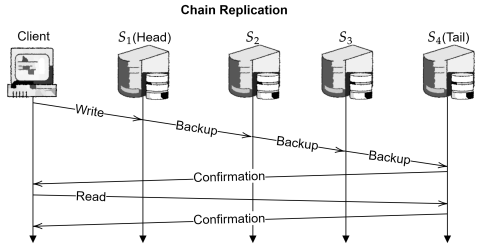
TaskPar. DataPar. PipePar. La&Ve.Clock:



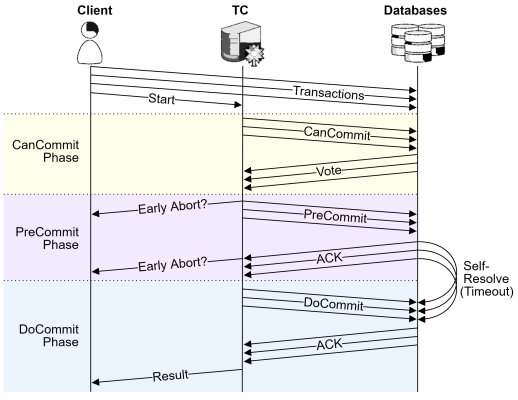
Snapshots: Consistent snap., captures causal dependencies, if $e_1 \rightarrow_r e_2$, and e_2 is in the snap., e_1 must also be present (otherwise it's inconsistent). If in the snap., e_1 sent a message to e_2, and e_2 is not in the snap., replay the message on snap. recovery. **Chandy-Lamp.Snap:** Alg. for capturing consistent snaps. Reqs: No failures, FIFO channels, Strongly conn. graph, Single initiator. (1) Marker sent to all out-chans., record local state. (2) On marker retrieval, block (empty) the chan. from which it came. Record local state, except for empty chans. Send marker to all out-chans. (3) Completion, When all processes have received and sent a marker, the snapshot is complete. (every processes' incoming channel is empty).—**Replication**— **Def:** Maintaining multiple copies of the same data across distinct nodes (machines), providing fault-tolerance, load-balancing, and availability. **Active vs. Passive Rep:** Active, client sends reqs. to all replicas, must process in FIFO order. Passive, client sends reqs. to one replica (primary), which propagates to others (backups). **State vs. Req. Rep:** State Rep., forwards the entire state to backups. Request Rep., forwards individual reqs. to backups. **Primary Commit:** Client→Primary→Backups→Primary→Client (Commit Point). **Arbitrary Serv. (CFG):** The Configuration Service Provider (CFG) ensures a controlled **failover** (switching to backups) in the event of a primary failure. **Chain Rep:** An ordered chain of s_n replicas, writes propagate from s_1 to s_n, where s_n reports back to the client. Reads speak directly with s_n. For any failover, the next adjacent successor takes over.—**Consensus** (2.3)—**State Machine:** Processes a seq. of inputs from a log, saving them in state. **Rep. State Machine:** Replicated logs across multiple machines, the processing of which is deterministic, generating the same state.**Consensus Model:** facilitates agreement of replicated logs between replicas. **Raft:** A consensus algorithm, with a central leader elected by the cluster in monotonically increasing terms. Liveliness is measured by periodic heartbeats, if the leader isn't reachable, followers, run for election. Split votes are dealt with the current candidates by timing out again (new timeout). Log Matching Property: Same index and term implies, same command and previous indexes are identical. Log Correction: The leader will find the index of the first mismatch, then overwrite the followers entries with its own. Leader Completeness: All proceeding leaders must have all committed entries of the previous leader. Leaders may only commit entries from their term. Candidates are rejected if they have a lower term, shorter log. Timings: $heartbeatReceipt \ll electionTimeout \ll failRate$. Cluster Reconfiguration: A joint consensus reconfiguration follows two phases: (1) propagate mix config of old and new, (2) propagate new config, which includes new servers or deletions. New servers enter with empty logs. Log Compaction/Snapshotting: Servers independently take snapshots, which truncates all committed logs, storing the last committed log index, term, and state (key-value pairs). Clients are always redirected to the leader. 2ReplaysSnap. Primary & Chain Rep.



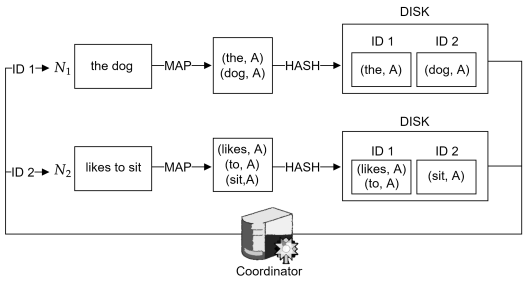


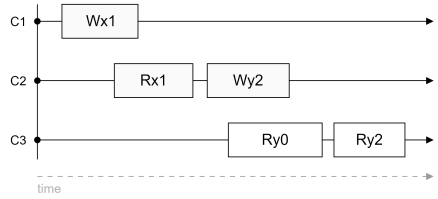


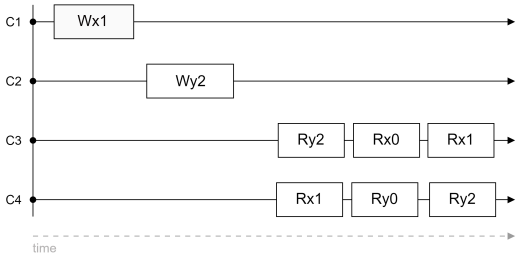
—**Failure Model Hierarchy** (2.4)—Crash-Stop: Process halts, cannot resume (undetectable) ⊂ Omission: Process fails to properly communicate ⊂ Crash-Recovery: Process halts, but can recover ⊂ Byzantine: Process exhibits arbitrary or malicious behavior.—**Consistency Models** (2.5.1)—**Def:** A distributed system's method on validating operation orderings on shared data. **Global Total Order:** Order of observable events agreed upon by all clients. **Strong Consistency (Str):** The client observes nodes agree on order execution (all node reads appear to be identical). **Weak Consistency (Wck):** The client temporarily observes that nodes disagree on shared data values. **Linearizability (Str):** Global Total Order with respect to real-time ordering (operation time intervals unmovable, but execution choice within them are). E.g., Raft leader-commit (happy-path) and Chain Replication. **Sequential (Str):** There is some Global Total Order found when shifting operation time intervals. **Causal Consistency (Wck):** Same as Sequential; However, clients may observe their own view on a Global Total Order. **Eventual Consistency (Wck):** Given no new writes, replicas will eventually agree on the same value after some time. **Causal Implications:** Linearizability $\implies$ Sequential $\implies$ Causal $\implies$ [First-In-First-Out (FIFO)/ Read-Your-Writes (RYW)]. **Release Consistency (Wck):** Push updates to all nodes after releasing the lock. **Lazy-release Consistency (Wck):** Push updates to the next node who acquires the same lock.—**Transactions** (2.6.1)—**ACID:** Atomicity, no partial effects, all or nothing. Consistency, A transaction takes the database from one valid state to another. Isolation, no interleaving of transactions. Durability, transactions once committed, are permanent even after system failure. **Serializability (Str):** Ensures the outcome of concurrent transactions is the same as if they executed in some serial (sequential) order. Differs from linearizability, which deals with real-time ordering of single tasks, as opposed to whole jobs. **Strict-Serializability (Str):** Serializability with real-time ordering; In particular, Serializability $\impliedby$ Strict-Serializability $\implies$ Linearizability. **Optimistic Concurrency Control (OCC):** Assumes conflicts are rare, proceeds without locking. (1) Prepare the transaction on all nodes, (2) Tell the coordinator to execute, validate the outcome, (3) Commit if Serializable, abort otherwise (isolation). **Timestamping OCC:** Helps with distributed OCC agreement on separated data, but aborts unnecessarily. **Two-Phase Commit (2PC):** Ensures **atomicity**, (1) Prepare Phase: After client's request is received on all nodes, the Transaction Coordinator (TC) sends a prepare message to all nodes, awaiting YES votes. (2) Commit Phase: If all nodes reply YES, the TC requests for all nodes to commit. If any fail to reply, the TC is blocked, sending the commit request to that node indefinitely (can't abort after this point). **3PC:** 2PC with an additional phase before the commit phase, ensuring people can commit (PreCommit Phase). If any nos or failure to respond in PreCommit, abort. If not partitioned, if the TC fails in the PreCommit phase, servers may attempt to reconcile with themselves about how to continue. **Pessimistic Concurrency Control (PCC):** Assumes conflicts are likely and prevents them by ac- quiring locks before any data access. It then preforms actions directly on shared data, ensuring serializability and isolation. The locks are released after the transaction is completed. 3PC.

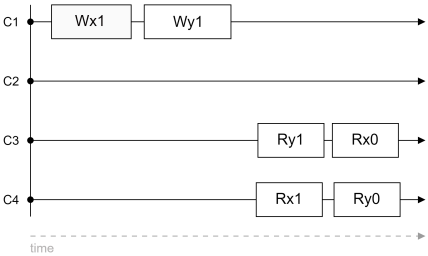


—**Distributed Shared Memory (DSM)** (2.8)—**Def (str):** A cluster which gives the illusion of a single shared memory space on a single machine, enabling multithreaded programs in a distributed setting. DSMs abstract away consensus and communication, opting to mimic virtual memory page primitives to communicate between nodes. **Sending Pages (Naive) (str):** Sending entire pages over the network is costly on bandwidth, but ensures consistency. **Sending Page Diffs (TreadMark) (Wck):** Keep a versioning history of every change, upon request, send the diffs to the page, avoiding false sharing (unnecessary updates). Versioning control is managed via vector clocks, which identify who made the changes (causal consistency). (1) Pages start as read-only (RO), which many may access. (2) When one claims read-write (RW) access, it invalidates all other copies. (3) Upon request, a page diff is sent over the network, the page now reverts to RO (at most one writeable copy). Weak consistency as per lazy-release style of data sharing.—**Virtualizing** (2.9)—**Sharding:** Splitting a dataset into smaller chunks, called shards stored on multiple nodes. Multiple copies increases safety. **Consistent Hashing:** Creates a hash ring of 2^m entries with keys of m-bit values. Servers are placed uniformly in multiple locations on a ring, serving for a range of keys. This helps with load balancing, as when one server goes down, the next clock-wise key ranges will help bare the load—**MapReduce:** (2.10)—**Def:** Map: Takes a set of input key-value pairs and produces a set of intermediate key-value pairs. Shuffle: this phase sorts the keys or hashes them (more efficient) into key bins that will be assigned to workers to reduce. Reduce: Takes an intermediate key and a set of values for that key, and merges them into a smaller set of values. Given W workers, M mapping tasks, and R reducing tasks, $W \gg N$ and $R \gg N$. Data is split up and given to workers in partitions. Map failure: Reassign the chunk, first one done is propagated. Reduce failure: Reassign the partition, they write to the same location (e.g., "/filepath/final_data/id"). Coordinator failure: Restart the entire MapReduce job. Failures are not recoverable, and are assumed to be rare. Slow workers (stragglers), will always be a bottleneck. MapRed. Lin. Even. ¬Caus.

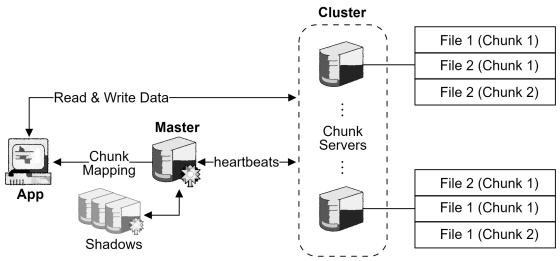






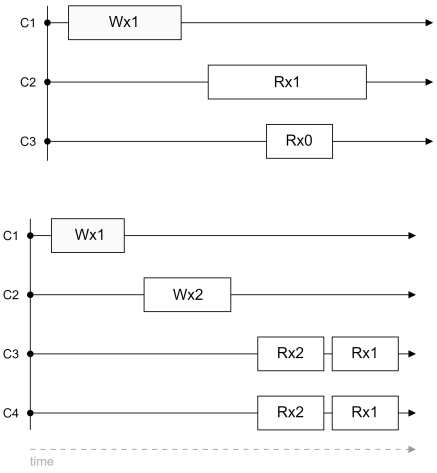


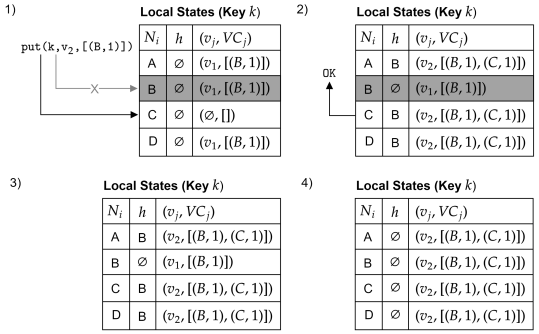
—**Google File System (GFS)** (2.11)—**Def (Wck):** Dist. file system, Scalable on cheap hardware, High availability on frequent failures (rep. need), High throughput of seq. reads and append-only writes. Files are split into 64MB chunks, given a 64 bit ID chunk handle. Chunks must be replicated N times (Typically $N = 3$). There is one TC, the master, which manages the file→chunk mappings and access control information. The client queries the master with (file name, chunk index), and receives the replica location. It caches this information to speak with the server directly. Clients may directly read from any chunk server (stale reads allowed). For writes, the client sends data to the closest replica, which then forwards it to the next closest replica. After all is done, the client orders the primary replica (picked by the master with a timed lease) to commit the data. The primary eagerly commits their own state and replies success, while the others lazily commit their state. The master will periodically check for stale chunks in case of failures (garbage collection through heartbeats). There are also background master replicas (shadows) that locally replay the master state in case of failure. The master has checkpoints, which are snapshots that truncates the log. Shadows also serve as read-only replicas to reduce load on the master. Shadows exhibit eventual consistency, slightly lagging behind the master. There is non-atomicity and non-serializability across chunks, but there is within a single chunk, due to the order of accepted writes from a primary on any given chunk. If a write is rejected as per full chunk, the client must refresh (find the new chunk server) from the master. Having a single master simplifies the design of the system.—**Dynamo: Key-Value store** (2.12)—**Def (Wck):** Always writeable (high availability), scalable, decentralized, low-latency performance (SLA: Service Level Agreement of 300ms on 99.9% of requests), Eventual consistency. Utilizes consistent hashing on 128-bit ordered list of N virtual nodes. Writes: client sends put(key, value), which is hashed to N_j, which serves as the coordinator/owner of the key. Such coordinator propagates in parallel to the next $N-1$ healthy nodes (preferred list). After W servers respond, the coordinator sends a success to the client. Reads: client sends get(key), which is hashed to N_j, who waits for R responses from the preferred list. Quorum Condition: $W + R > N$, ensures sufficient overlap, mitigating stale reads. Failure: If k hashes to A's segment, we retry the next healthy node. We hint (notify) the next node B that A is down. B then stands in as the coordinator, with the addition of hinting A to the cluster (Sloppy Quorum). During runtime, all N nodes gossip with each other, sending heartbeats at random. During this phase, if A recovers, a random server C may notice this, and preform a hinted-handoff, passing along the operations A missed. Diverging data is resolved via a versioning vector clock for each object (key, value), where each cell is a node, and the value is a monotonically increasing counter of its participation. During reads, the coordinator attempts to resolve the vector clock, if it can't, it returns all versions to the client. The client must decide how to merge the data (often unioning them in case of two carts). Then sends the merged data back as a put operation. In the background, nodes will try to resolve stale data via merkle trees (hash trees). Merkle trees, first hash a range of keys (leaves), then combines two hashes into a parent (intermediate nodes), until they reach the root node (summarizes the entire tree). Nodes use this to efficiently find divergent data. GFS.

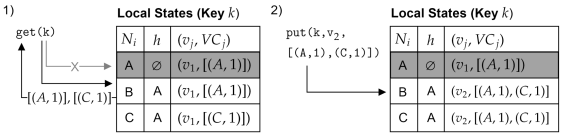


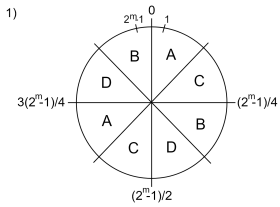
—**Spanner** (2.13)—**Def (Str):** A globally distributed database, linearizeble reads and writes across shards (globaly agreed timestamp), High availability, low-latency, lock-free read-only transactions, strict serializability, with atomic transactions. Utilizes Multi-Version Concurrency Control (MVCC): Each write transaction assigned a uniquely increasing timestamp with every key-value pair. Safe Time Rule: Before transaction T reads key k, it must see a timestamp greater than the last write to k (if any). Atomic clocks are used to mitigate clock drift, and are synchronized via GPS satellites. The TrueTime API, provides a `TT.now()`, which returns a $[t_{earliest}, t_{latest}]$ interval, where the real time must be between the two (inclusive). The difference of the two intervals is the uncertainty δ, which is the time it must wait before reads or writes can be performed. The protocol uses paxos (consensus model like raft) to ensure fault-tolerance and availability of replicas. Read-write Transactions: two-phase locking (2PL) and two-phase commit (2PC) are used to ensure isolation and atomicity respectively. The first phase acquires locks on needed keys and reads the values, then 2PC commences with one of the shard paxos leaders as the TC. After the commit, the locks are released.—**TLA+** (2.14)—is a high-level specification language based on temporal logic that lets you model distributed algorithms, exhaustively explore all possible states, and prove safety and liveness properties. While it excels at uncovering subtle bugs and proving correctness, TLA+ specifications do not translate directly into production code—implementing a verified design remains a substantial engineering effort.

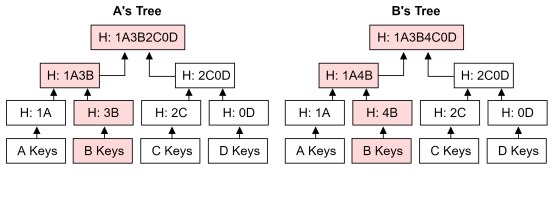
CHash. Caus&Seq. HintOff. DynCliRec. Merkle.











δ-time. 2PL2PC.

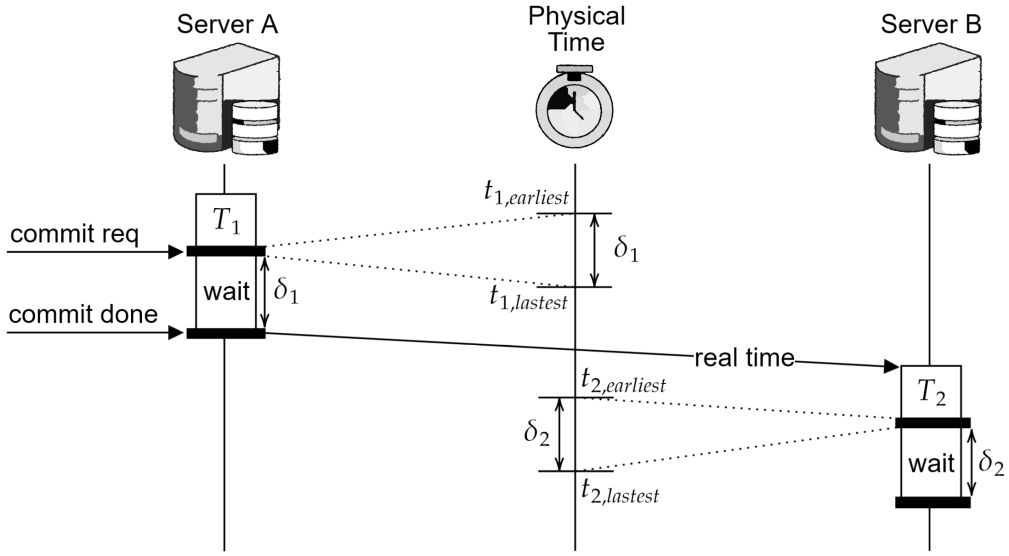


Figure 3.1: Upon receiving a commit request, each coordinator (e.g. A) calls `TT.now()` and obtains an uncertainty interval $[t_{1,\text{earliest}}, t_{1,\text{latest}}]$. It then delays for $\delta_1 = t_{1,\text{latest}} - t_{1,\text{earliest}}$ before finalizing its commit timestamp.

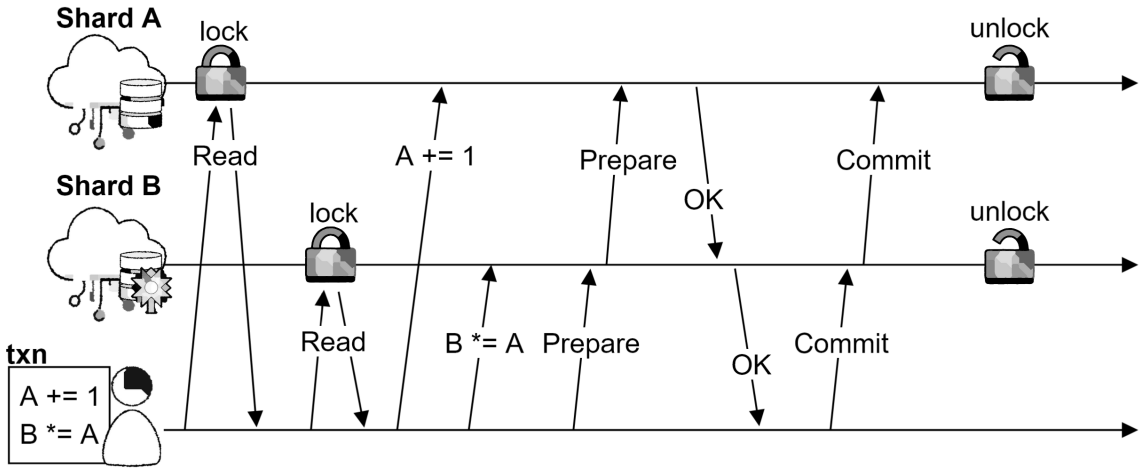


Figure 3.2: A Read-Write Transaction 2PC Commit in Spanner, where txn is the transaction, B's Paxos leader is the coordinator, and A another participant. First, reads acquire locks, then 2PC begins. Commands are sent, and then the `PREPARE` and `COMMIT` phases begin, ending with the release of locks.

Bibliography

[1] Neso Academy. Virtual memory: Multilevel page tables, May 2020. Accessed: 2025-04-19.

[2] Yair Amir. Performance of multilevel paging. https://www.cs.jhu.edu/~yairamir/cs418/os5/sld035.htm, 2003. Slide 35, Operating Systems Course (CS 418), Johns Hopkins University.

[3] Computerphile. But, what is virtual memory?, 2023. Accessed: 2025-04-18.

[4] Ritwik (https://cs.stackexchange.com/users/58645/ritwik). calculate the effective (average) access time (e at) of this system. Computer Science Stack Exchange. URL:https://cs.stackexchange.com/q/69017 (version: 2019-04-27).

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

[6] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[7] Tim Roughgarden and Gregory Valiant. CS168: The Modern Algorithmic Toolbox, Lecture #1: Introduction and Consistent Hashing. Lecture notes, Stanford University, April 2024.

[8] ScyllaDB. Consistency models definition, 2025. Accessed: 2025-03-25.