DISTRIBUTED COMPUTING FROM FIRST PRINCIPLES

By

Kenneth Emeka Odoh

https://kenluck2001.github.io

May 2025

Preface

”I would maintain that thanks are the highest form of thought; and that gratitude is happiness doubled by wonder.” – G.K. Chesterton

”An adventure is only an inconvenience rightly considered. An inconvenience is only an adventure wrongly considered.” – G.K. Chesterton

This work is motivated by my quest to acquire comprehensive knowledge in a technical field with practical real-world applications. Our manuscript aims to inspire the next generation of software practitioners (Engineers, scientists, and physicists) to apply distributed computing paradigms to address their challenges.

I am grateful to the numerous reading groups in Vancouver that spurred my interest in Distributed Systems. Despite my humble beginnings, I am now privileged to have developed into a seasoned Software Engineer. This book represents my opportunity to contribute back to society. Writing this book has been the most challenging endeavor of my evolving career. Hence, this work is a **charitable undertaking** and will be **royalty-free** to maximize the benefit for underrepresented minorities.

While living mentors are invaluable in my career development, these late individuals had the greatest influence on my life: Thomas Aquinas (bright star of the Dominican Order), Chinua Achebe (chronicler of Igbo civilization), G.K. Chesterton (prince of paradox), and Tupac Amaru Shakur (thug poetry). In the words of G.K. Chesterton, ”A good novel tells us the truth about its hero; but a bad novel tells us the truth about its author.” Therefore, the judgment rests with the readers [1] [2] .

[1] The motivation for this book is detailed at `https://kenluck2001.github.io/blog_post/authoring_a_new_book_on_distributed_computing.html`.

[2] The front page illustration was created by Rofiat Ibrahim using pencil on paper. Rofiat is a talented artist and a student whom I met by chance at Yaba College of Technology in Lagos, Nigeria, in December 2022. Her excellent drawing features a broken kola nut and a set of palm wine jars, items of profound cultural significance in Igbo tradition. This figure crystallizes my thought that knowledge should be ”**free as in free beer**”.

Acknowledgement

”When it comes to life the critical thing is whether you take things for granted or take them with gratitude.” – G.K. Chesterton

Archit Goyal reached out and provided source code on CRDT among other major reviewing and coding contributions. Archit is a Senior Software Engineer at LinkedIn, deeply involved in building and productionizing large-scale distributed data systems. He is one of the early engineers behind LinkedIns Flink Batch platform, powering Ads AI workloads and high-throughput, low-latency pipelines. Much of his work focuses on scaling stateful computations, disaggregated shuffle, checkpointing trade-offs, failure recovery, resource fairness, and multi-cluster observability.

I want to thank Anselm Eickhoff and Charles Krempeaux, who were responsive to my questions about CRDT. Also, I would also like to thank Harrington Joseph, who suggested that I focus on gaining a low-level understanding of parallel programming. We provided credits for both anonymous reviewers that provided tangible support in making this manuscript.

Kindly make a free-will donation to support my work, including open-source development, blogging, research papers, or textbooks. Thank you so much for your financial support.

Donate: `https://buymeacoffee.com/kenluck2001`

Contents

Chapter 1

Introduction

"What we know is a drop, what we don't know is an ocean." – Isaac Newton

Have you ever wondered how a typical distributed system works under the hood? Are you looking for a pedagogical guide with complete implementations and tricks of the trade? Look no further and read my writing on the topic. We have implemented several foundational algorithms in Distributed Computing. This paradigm has become ubiquitous in the industry where multiple systems interact to solve computational tasks, with applications including distributed stores (databases), IoT sensor networks, and the Internet. The exponential growth in deploying Distributed systems for solving real-world problems has led to a resurgence in understanding Distributed Computing from scratch [1] . Consensus is a foundational requirement for distributed systems, allowing disparate nodes to synthesize a consistent global perspective from their local operations. This collaborative agreement forms the very objective of distributed algorithms.

Distributed systems operate in inherently uncertain and chaotic environments, facing network unreliability through intermittent connectivity, message loss, and out-of-order delivery. Therefore, fault tolerance must be a fundamental architectural requirement, not an afterthought. As a result, managing these complexities demands rigorous correctness guarantees to prevent unintended behaviors at scale across multiple nodes and unreliable channels. In production-grade software, formal verification is

[1] My research in Distributed Systems is a two-phased process: The first phase was covered in the `https://kenluck2001.github.io/blog_post/distributed_computing_from_first_principles.html`. The second phase is the current book that you are reading. A multi-phased approach to research has helped to minimize risk.

crucial to reduce bugs and guarantee safety and liveness properties are never violated, especially in mission-critical applications.

A few years ago, my deficiency in the low-level knowledge required to build a large-scale distributed system without utilizing third-party packages motivated me to research this topic from first principles. I subsequently became a better Software Engineer in the process. My initial foray into distributed systems began by participating in a Distributed System meetup in Vancouver, where I eventually became the organizer. Our study group utilized materials from the Distributed Systems course at KTH Sweden. Our goal was mainly to gain intuition and theoretical knowledge on the subject. Fortunately, I took a further step by completing the Parallel, Concurrent, and Distributed Programming in Java Specialization on Coursera, where OpenMPI was mentioned. I recognized this as the missing link in my exploration of implementing low-level distributed algorithms.

As I started researching, I noticed a troubling trend in Distributed Systems research: the most significant work in the field tends to be done in top research labs, advanced technical schools, and by seasoned open-source contributors. Unfortunately, this status quo is unacceptable. Hence, I am motivated to invest in this research to distill this knowledge to a diverse audience. Our focus is on delivering scientific content without diluting its quality and maintaining world-class rigor. We aim to create disruptive change by sharing knowledge hidden in plain sight. One axiom from the Zen of Python posits that ”practicality beats purity,” so rather than pontificate on the state-of-the-art distributed algorithms, we have adopted the approach of solidifying the fundamentals.

Notes:

1. This work is based on non-proprietary content unrelated to my employer. The blog summarizes the first phase of my two-phased research on low-level Distributed systems. The second phase is completed in this textbook.

2. All source code is the original work of the author.

3. Unlike other intellectual works, some of our references will be secondary sources. This decision is taken to keep our references within a reasonable scope.

Full disclosure: I implemented portions of the KTH Distributed system course that ran on Edx, taught by Professor Haridi. I am grateful as it was my first exposure to

Distributed computing. I will also include some of his slides (Paxos, sequence Paxos) in this book with full attribution.

Terminologies

1. Quorum: a set containing a majority of correct processes.
2. Nodes, replicas, and processes will be used interchangeably and have the same meaning within the context of this book.
3. Model: an abstraction of the dynamics of a system.
4. Reliable broadcast: a message sent to a group of processes is delivered to all or none.
5. Atomic commit: processes either all commit or all abort a transaction.
6. Transaction: a series of operations that must be completed without interruption. If any operation fails, the entire transaction is aborted, e.g., 2-phase commit, 3-phase commit.
7. Multicast: a single source sends to multiple destinations and operates in LAN and WAN environments.
8. Broadcast: a single source sends to multiple destinations and operates in a LAN environment.
9. Computational graph: an abstraction of the ordered sequence of processes.
10. Optimistic concurrency is a suitable strategy when low contention is expected. Computation on a shared object is expensive compared to the overhead of locks in such scenarios.
11. RAID: This storage paradigm, known as a Redundant Array of Independent Disks, provides fault tolerance in the face of disk failures.
12. Volume: This refers to the available storage space accessible by the operating system.

1.1 Algorithm Implementations

My source code was tested with the following specifications:

Setup	Version
Operating system	Ubuntu 16.04.6 LTS
GCC	5.4.0 20160609
OpenMPI	1.10.2
Python	2.7

Here is my list of implemented Distributed algorithms with associated source codes. They include:

S/N	Algorithms	Source codes
1	Logical clock	vector2.c, lamport1.c
2	Paxos (with Snapshot)	single-paxos3-snapshot.c
3	Sequence Paxos	sequence-paxos4.c
4	Failure detector	failure-detector.c
5	Leader election	leader-election3.c
6	Grow-only Counter CRDT	crdt-gcounter.c
7	Last-Write-Wins Map CRDT	lww-map.c
8	Raft	Raft algorithm in Subsection 4.3
9	Distributed shared primitives	shared primitive in Subsection 8.4.2
10	Distributed hashmap	lamport1-majority-voting8.c
11	Two-Phase Commit	two-phase-commit.c
12	Autoscaling Framework (Serveless)	autoscaling.py in Subsection 8.4.1

Source Code: click here

Some of my implementations are in the playground folder, where you can follow my thought process during incremental development. Some of the implementations in the playground contain faulty solutions due to incorrect assumptions. However, the implementations referenced directly in this book are verified by the author and relatively free of obvious errors. We settled on OpenMPI as the underlying library to provide low-level functionality. This choice is preferable to using a Remote Procedural Call, which obscures the message-passing paradigm by passing arguments as messages. Although conceptually similar, MPI provides a structured way of implementing our

Distributed algorithms. Hence, all implementations will strictly use OpenMPI in C. Event-based programming based on message passing serves as a powerful abstraction for building Distributed systems. Therefore, we are using this approach for all of our implementations. For the sake of pedagogy, we have omitted tracing and invariant proofs. While these are important in Distributed systems scholarship, we anticipate that users will learn by doing and not be hindered by excessive mathematics. However, for those who wish to focus more on writing proofs, we recommend this resource [33].

1.2 Overview

This book is organized into a number of Chapters. We begin by providing motivation, refreshers on prerequisites, foundational theory and algorithms, formal verification, and exercises.

Chapter 1 motivates embarking on the adventure of understanding the underpinnings of Distributed computing from scratch. This section includes a list of implemented algorithms and sets the stage for the entire book.

Chapter 2 provides a quick primer on parallel programming and network programming concepts that are fundamental for forming the correct mental model required to understand more advanced concepts in the book. It is in this section that we introduce OpenMPI as the message-passing library to support our implementation of Distributed algorithms.

Chapter 3 describes Distributed systems. It also discusses the CAP theorem, the FLP impossibility of consensus, the Two Generals' Problem, accounting for the absence of a global clock through the introduction of logical clocks, and a principled way to detect failures.

Chapter 4 describes foundational Distributed algorithms such as Paxos, Raft, election algorithms, and stabilization algorithms.

Chapter 5 describes anti-entropy techniques. This is critical for achieving a consistent state across nodes, as divergences can affect agreement in these systems. The approach focuses on determining divergence across replicas, a fault-tolerant way of propagating changes to enforce a consistent state across replicas, reducing the time needed for eventual consistency, and performing error correction.

Chapter 6 describes peer-to-peer networks. This helps to create fault-tolerant systems that do not depend on a single server, as central servers are a source of single points of failure. Building peer-to-peer networks can help produce the redundancy needed to achieve a system where a single server is not a bottleneck.

Chapter 7 describes the formal verification of Distributed systems. It is imperative that in mission-critical systems, safety and liveness properties should not be violated at any time.

Chapter 8 describes topics such as distributed commit, coding philosophy, case studies, practical considerations for deploying distributed systems in the wild, testing strategies, evaluation metrics, and exercises.

Chapter 2

Prequisites For Distributed Computing

"He who knows all the answers has not been asked all the questions." – Confucius

"Anything worth doing is worth doing badly" – G.K. Chesterton

We have demonstrated utilizing the message-passing paradigm for implementing Distributed Systems in this work. The requirement for nodes to exchange information over a channel, warrants a solid understanding of network programming to grasp packet dynamics as information flows across the channel. Networking programming skills are closely complemented with the knowledge of Parallel programming. As a result of these prerequisites, we provide a brief introduction to both subjects in this manuscript. This knowledge is sufficient to understand the internals of real-world Distributed systems, as covered in Sections 2.1 and 2.2.

2.1 Network Programming

The socket is the fundamental file descriptor for networking. Recalling the Unix philosophy that everything is a file, the 'socket()' function returns a descriptor. Sending a message over a network becomes analogous to writing to a file stream, while receiving a message is comparable to reading a file stream. However, in this context, the network acts as the channel for this communication. There are two basic socket types, as outlined by [27]:

- Stream socket ('SOCK_STREAM')
- Datagram socket ('SOCK_DGRAM')

Most local networks utilize internal IP addresses, whereas Internet-facing gateways employ external IP addresses with Network Address Translation (NAT) performing the conversion from the Internet IP to the internal IP.

A server typically operates in a listening mode. Clients send messages to the server to trigger actions, resulting in a response to the caller. Socket programming is a method for machines to connect over a network. This communication occurs via a protocol using a set of predefined conventions that enable parties to understand message structures, codebooks, and other auxiliary information necessary for the synthesis of exchanged data.

Network Protocol

Network protocols support varying degrees of error recovery, fault tolerance, and overhead rates for message transfer. The design requirement should inform the choice of the network protocol.Be it Transmission Control Protocol (TCP) [1] , User Datagram Protocol (UDP) [2] , Quick UDP Internet Connections (QUIC) [3] , or Licklider Transmission Protocol (LTP) [4] respectively.

Transmission Control Protocol (TCP) is a connection-oriented protocol operating at the transport layer. This protocol requires a dedicated connection must be established between the sender and receiver before data can be exchanged. A key characteristic of TCP is its provision of congestion control mechanisms, which dynamically adjust the transmission rate to avoid network overload. Furthermore, TCP guarantees reliable and order-preserving delivery of packets, ensuring that all data arrives at the destination correctly and in the order in which packets were sent. However, these features contribute to TCP being considered a "heavyweight" protocol due to the overhead associated with connection management and reliability mechanisms. Error recovery gets handled by the protocol, where lost messages are re-sent. The TCP protocol described in Figure 2.1.

[1] https://datatracker.ietf.org/doc/html/rfc9293

[2] https://datatracker.ietf.org/doc/html/rfc768

[3] https://datatracker.ietf.org/doc/html/rfc9002

[4] https://datatracker.ietf.org/doc/html/rfc5326

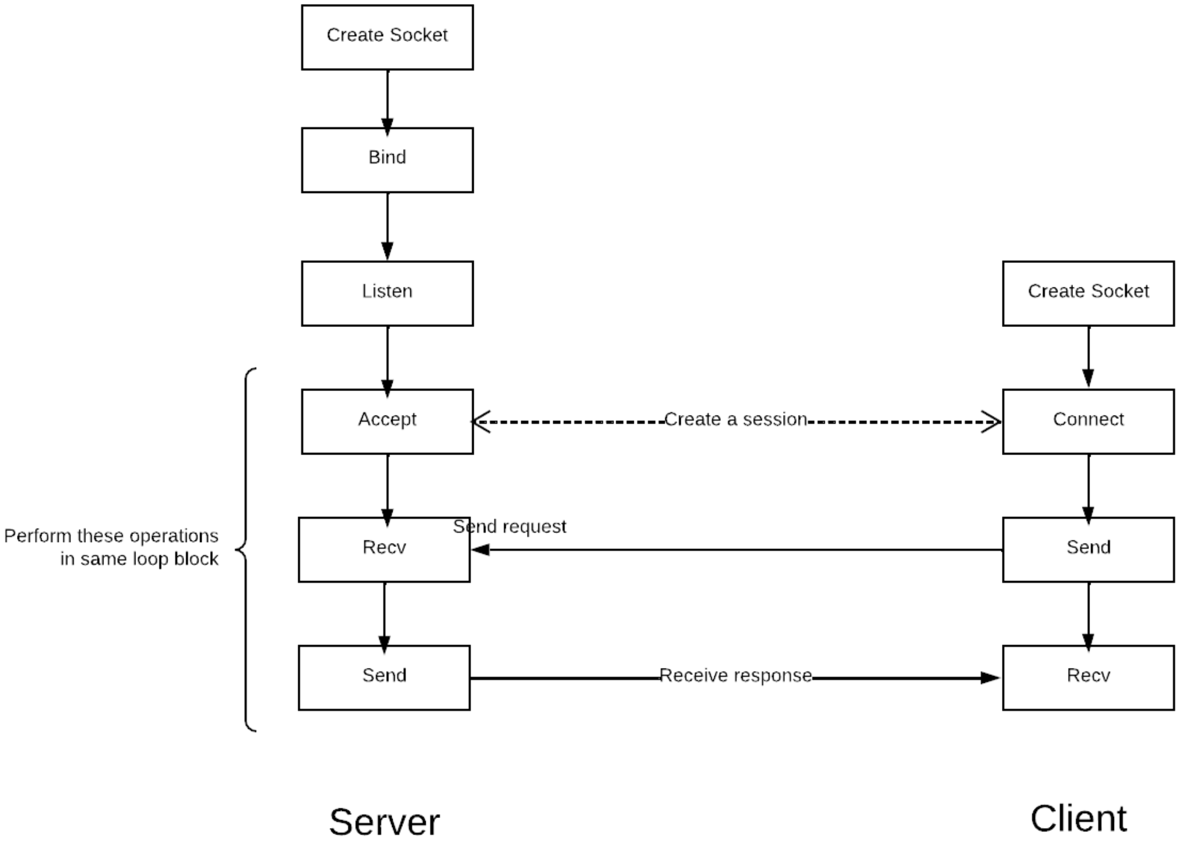


Figure 2.1: Sequence diagram for TCP.

User Datagram Protocol (UDP) is a connectionless-oriented protocol, that operates at the transport layer. Unlike TCP, UDP does not require a established connection before communication begins. Consequently, it is lacking the built-in congestion control and reliability guarantees of TCP, meaning packets may be lost, duplicated, or arrive out of order. The absence of these features makes UDP a "lightweight" protocol with lower overhead, making it suitable for applications where speed and low latency gets prioritized over guaranteed delivery. Error recovery is handled by secondary protocols implemented on top of UDP to provide resilience. The protocol gets described in Figure 2.2.

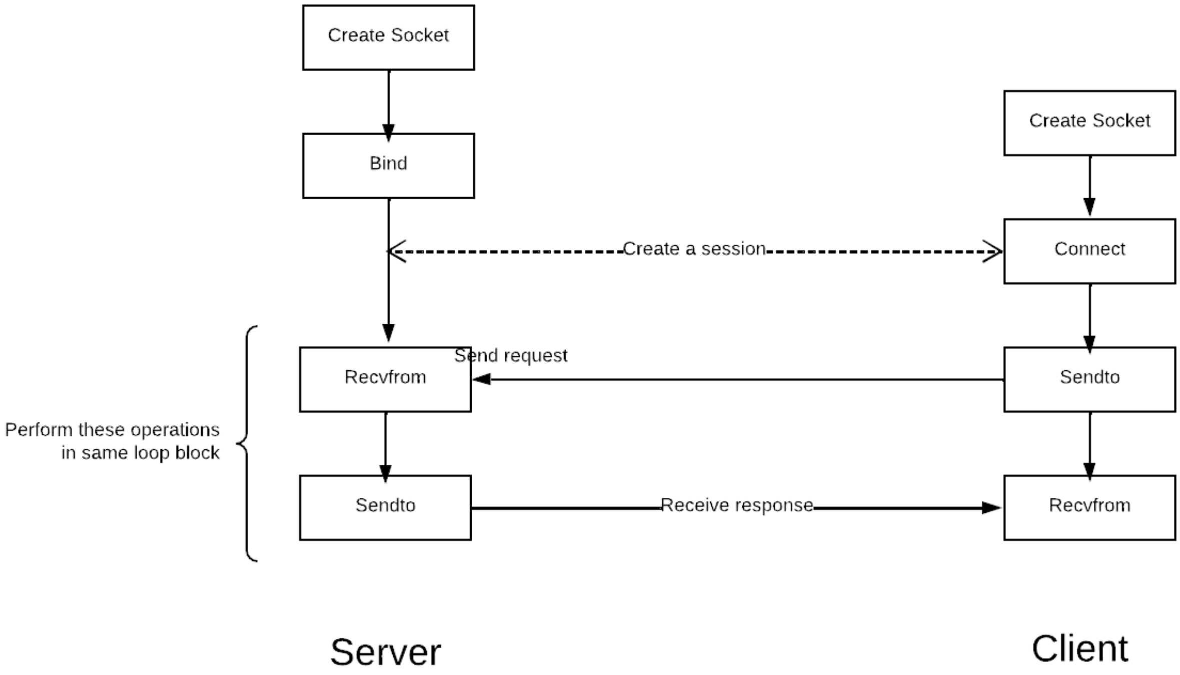


Figure 2.2: Sequence diagram for UDP.

Quick UDP Internet Connections (QUIC) represents an evolution of UDP. While built upon UDP, QUIC incorporates mechanisms to provide reliability and security features associated with TCP, such as reliable, in-order delivery and encryption. A significant advantage of QUIC is its ability to achieve these guarantees with reduced connection establishment time and lower overhead compared to TCP, making it a more efficient option for many applications.

Licklider Transmission Protocol (LTP) is a lightweight protocol that operates at the data link layer, a lower layer in the network stack compared to TCP and UDP. LTP is designed for challenging network environments characterized by intermittent connectivity and long delays, such as deep space communication. It does not implement congestion control and allows for opportunistic message transfer, where sending might be delayed until a suitable communication channel becomes available. LTP is a hybrid protocol that can be configured to provide reliable or unreliable delivery depending on the requirements of the application.

The application requirement should guide the choice of the appropriate network protocol in the design of a Distributed System. For more information, refer to the illustration in this blog.

When communicating between nodes in Distributed systems, a channel is necessary to provide a medium for message transfer. The types of channels in use in Distributed systems [37] include:

- Fair loss link
- Stubborn link
- Perfect link
- Logged perfect link

Each of these links has specific and unique characteristics. For more information on this subject, please consult the following resources:

- UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API By W. Richard Stevens, Bill Fenner, Andrew M. Rudoff
- Beej’s Guide to Network Programming Using Internet Sockets by Brian Hall

2.2 Parallel Programming

In Unix, ‘fork()‘ creates a new process, where a parent process creates a new child process. The parent process must wait for the child process to exit, allowing the child process to terminate properly. This behaviour can become problematic if the child process becomes defunct (a zombie) and the parent process ignores waiting. In some systems, the ‘init‘ process reaps (destroys) defunct processes. A child process becomes a zombie until the parent process waits or the parent ignores the ‘SIGCHLD‘ signal [27, 28]. Ignoring the waiting for the child process to exit in the parent process:

```
int main()
{
 signal(SIGCHLD, SIG_IGN); //don’t wait
 fork();
}
```

Processes and threads are core foundations for building Distributed computing systems [9, 12]. Processes are units of work distribution. Threads are units of concurrency. Parallel programming presents significant challenges, including data sharing, coordination, deadlock, lock granularity, and others [9, 12]. It is possible to have multiple threads within a single process. Some benefits of this include [9, 12]:

- Memory and resource efficiency due to sharing.
- Responsiveness (no network delays within the process).
- Performance (increased throughput); note that if the process blocks, every internal thread blocks as well.

Another combination is to have multiple processes within a node. Some benefits of this include [9, 12]:

- Responsiveness (mitigating JVM delays, if applicable).
- Scalability.
- Availability and fault tolerance.

There are different forms of parallelism (task, functional, loop, data-flow) [9]. Java's popular fork-join framework is based on the divide and conquer paradigm, which is useful, where the problem can be decomposed in this sub-problem structure [9].

When using threads or processes, it is ideal to utilize the optimal number to prevent reduced performance due to degradation from increased load (due to process isolation) and communication costs for sharing information between processes.

Promises and futures are a popular form of parallelism where a deferred call is made while simultaneously performing another disjoint task in simultaneously parallel, and then joining on the result from the main thread.

Guaranteeing reproducibility can be a desirable goal. Quasi-randomness may be acceptable in some contexts. A parallel program can exhibit the following characteristics to address this behavior [9]:

- Functional determinism: the same input to the same function always yields the same output.

- Structural determinism: any repetition of the parallel program yields the same computational graph.

Let's say that your process is processing permutation-invariant data (e.g., unsorted data); in this case, only functional dependency is needed, not structural determinism. This characteristics can provide clever shortcuts to optimize your program.

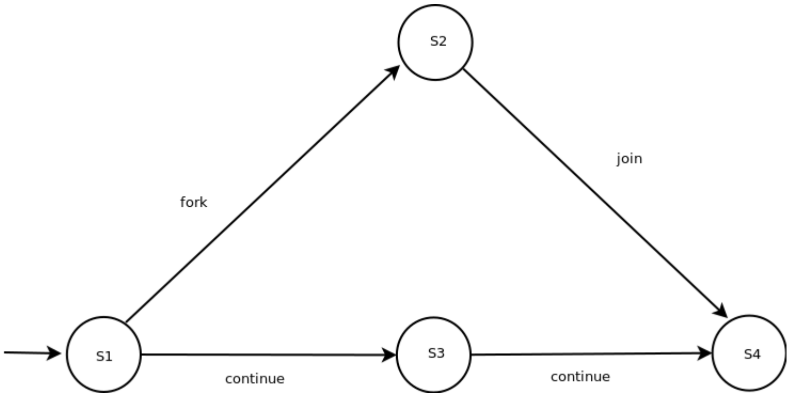


Figure 2.3: An example of a computational graph.

- Each node represents a sequential or sub-computation step.
- Each edge represents an ordering constraint.

Without a fork or join operation, the computational graph shows a straight line from the start to the finish nodes. In our example, S2 and S3 runs in parallel, as shown in Figure 2.3.

Metric performance:

- Work: the sum of execution times across every node.
- Span: the length of the longest path (critical path length).

For fork-join programs, the edges can be:

- Continue edge: captures the sequencing of steps within a task.
- Fork edge: connects a fork operation to the first step of a child task.

Ideal parallelism (*ip*) is given by: $ip = work/span$ For a sequential program, the ideal parallelism is 1. This requires having a computation graph [9]. Unfortunately, we don't always have a computation graph. In such cases, we can use Amdahl's law, which states that the speedup of a threaded program is limited by the sequential portion of the computation across all processes [9, 12]. There are also alternatives like share-nothing architecture (using the message-passing paradigm), e.g., actor model, and distributed actors, which make use of message passing. Non-blocking approaches using 'getAndAdd()' and 'compareAndSet()' are also available [9].

Properties of a Parallel Program [9]

- Safety: bad things never happen.
- Liveness: good things eventually happen.

Let us use the analogy of a traffic stop:

- Safety: vehicles can only move in one direction at a time.
- Liveness: even with traffic, every vehicle is guaranteed to eventually leave the junction.

2.2.1 Process Communication

Fine-tuning control of computer programs involves mutual exclusion and synchronization. Let us define these two concepts:

Mutual exclusion: two or more events do not happen at the same time, e.g., event A precedes event B.

Synchronization guarantees the order of access of multiple threads to a shared resource. Synchronization outside a computer can be verified in real life by a clock. However, achieving computer synchronization without a clock is challenging, as there are also issues with the accuracy of clocks due to drifts. It is possible to do so using computational graphs, especially in a parallel program. While it is easy to know the order of execution within a process, it is challenging to ascertain the order of execution in a threaded program. Synchronization is achievable by signaling or instructing the other party to wait until a condition gets met. There are traditional synchronization problems posed in the literature with wide-applicability. These include the producer-consumer problem (with multiple variants), the reader-writer problem, the starvation problem (bounded wait on a semaphore), and the dining philosophers problem [9, 23, 28].

A barrier is a construct that blocks multiple threads and releases every blocked thread upon the arrival of the last one [23]. This paradigm is commonly known as a phaser in Java and is widely employed in MPI for creating collective operations. This construct can also be used for pipelining [9, 23, 28]. A good book on interprocess communication is Beej's Guide to Unix IPC by Brian Hall. There are multiple venues for sharing information between processes in Unix.

- Signal: A signal is a way of performing process communication. One process raises a signal, and another process delivers it to the destination handler to effect a custom callback. Its thread-safety and interrupt-safety are uncertain.

- Pipe: This is the simplest form of process communication, where a process writes to one end of a pipe and reads from the other end. There are many variations, such as FIFO (named pipe).

- Message queue (‘msgsnd()‘, ‘msgrcv()‘): see usage details.

- Semaphore: At initialization, a semaphore is set to a user-defined value, representing the number of threads that can pass through the critical section before blocking. Threads can increment or decrement the semaphore's value, which cannot be read outside the semaphore construct. If the semaphore's value becomes negative, every thread will block unless it increments the value to be greater than 0. Most semaphore implementations are not atomic, and race conditions may occur if multiple threads access the resource simultaneously. One way to mitigate this issue is to have a single initialization process create the semaphore before the main processes begin to run. The main processes can then access the semaphore but cannot create or destroy it. There are similarities to restricting access using locks, permissions, etc.

- Shared memory segments: A process writes to a memory segment, and another process reads from the same segment. Coordinated access to the segment still requires using locks or semaphores (e.g., ‘shmget()‘, ‘shmat()‘).

2.2.2 OpenMPI

OpenMPI [6, 7, 8, 10] is an implementation of the message-passing paradigm for the exchange of information between processes, widely used in rendering farms, high-performance computing, and Scientific computing. For example, Curie, a French supercomputer, makes extensive use of OpenMPI for processing workflow. One of the best resources for learning MPI on the Internet can be found on OpenMPI website.

The number of processes, *size*, is determined during the setup of the OpenMPI communicator. However, adjustments can be made using sophisticated process group management. Every process is assigned a rank, ranging from 0 to *size*-1.

- mpi_bcast: all processes must wait until all processes have reached the same collective.
- mpi has optimized the implementation of distributed operations.

Two types of communication are in use:

- Point-to-Point: two processes in communication.
- Collective: every process is communicating together

Point-to-Point

- send: move data from one process to another process.
- recv: accept data sent to the process by other processes.

Collective Operation

- broadcast: one process sends a message to a group of processes.
- reduction: one process gets data from every other process and applies transformation (sum, minimum, maximum).
- scatter: a single process partitions the data and sends each chunk to every other process e.g. MPI_Scatter.
- gather a single process assembly data from different processes in a buffer e.g MPI_Gather. MPI_AllGather is to gather and scatter the results from every process. There are synchronization primitives like locks, barrier

One-Sided Communication

MPI allows for remote memory access (RMA). Here are some commands which include:

- MPI_WIN_create()
- MPI_WIN_allocate()
- MPI_GET()
- MPI_PUT()
- MPI_Accumulate()
- MPI_Win_free()

Unlike the two-sided communication model, which requires a sender and a receiver for data transfer, the one-sided communication model allows a process to directly access another process's memory space. Consequently, only one process initiates communication directly, minimizing data transfer and CPU intervention. However, there is a caveat that the ordering of RMA operations cannot be safely guaranteed [6, 10]. In our implementation of Distributed shared primitives, we made extensive use of one-sided communication. While RMA is inherently non-blocking, we are currently employing it in a blocking mode using locks. Implementing an epoch with MPI_Win_fence might be a more suitable approach in our logic to fully leverage the non-blocking features. MPI_Compare_and_swap could also present a better alternative [6, 10]. If one-sided communication is used in place of the default two-sided communication, the concept of explicit acknowledgments (ACKS or receipt confirmations) as in two-sided communication doesn't directly apply. The communication is more akin to direct memory manipulation. Here is an example of an epoch:

```
MPI_Win_fence    // start of epoch
.
.
.
MPI_Win_fence    // end of epoch
```

Is using a fence, better than using locks on the level of concurrency granularity? Hints, MPI_Win_fence is a collective using ideas from barriers.

Best Practices

- No user-defined operation in MPI_Accumulate.
- Ensure local completion before accessing the buffer in an epoch.
- It is impossible to mix MPI_GET, MPI_PUT, MPI_Accumulate in a single epoch

Benefits

- It can help to reduce synchronization.
- It minimizes data movement (excluding buffering).

Chapter 3

Basics of Distributed Computing

"Without education, we are in a horrible and deadly danger of taking educated people seriously." – G.K. Chesterton

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

Distributed Systems [9, 12] are sets of nodes (devices) connected by communication links that operate as a single, cohesive system. Although each device functions locally and independently, the overall system appears as a unified global entity, providing the "single view illusion." Examples include the Internet, edge computing environments, mobile networks, and sensor networks, as illustrated in Figure 3.1.

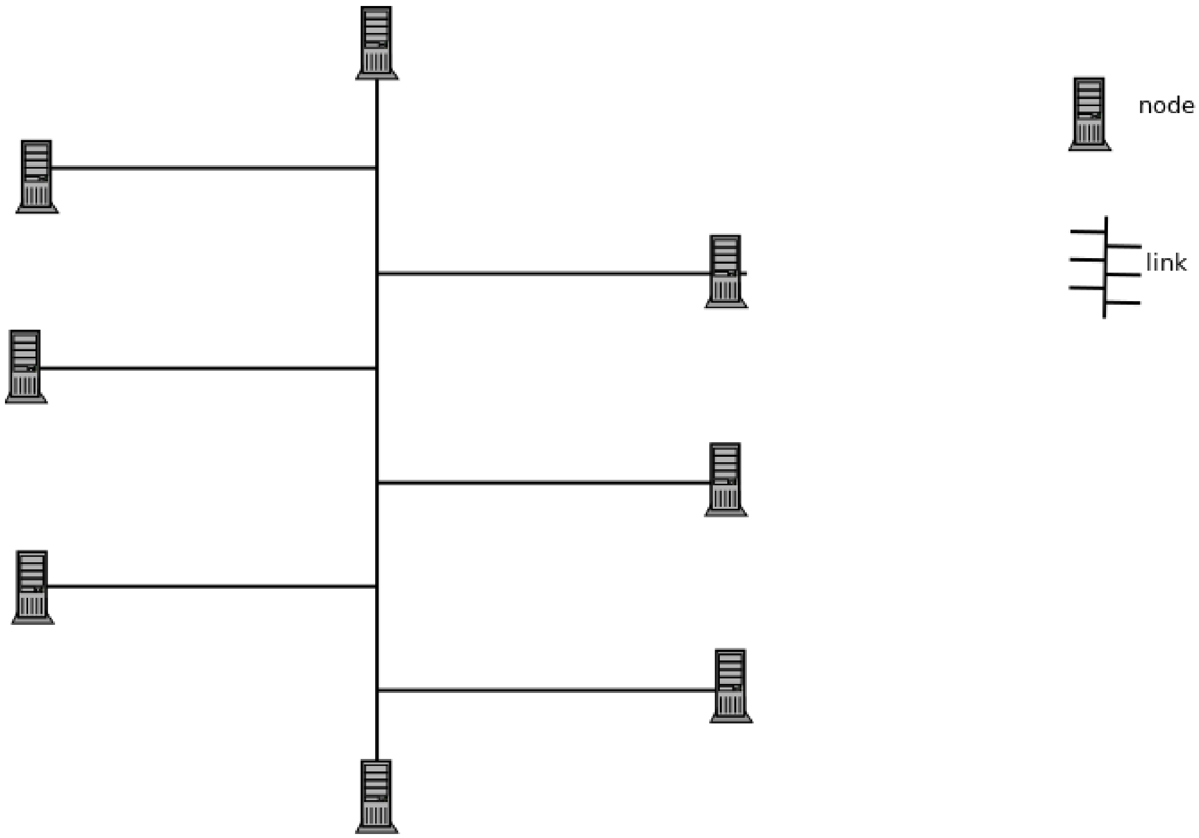


Figure 3.1: Network of nodes in a cluster.

The benefits of employing a Distributed system can include:

- Fault tolerance (reliability)
- Scalability
- Load distribution

A common misconception is that a Distributed system inherently possesses scalability. However, this is not always the case. Horizontal scaling, which involves adding more nodes, yields performance gains primarily when the communication cost between nodes is low and the task at hand is parallelizable and can be effectively distributed. The challenges inherent in distributed systems often involve partial failures (such as network or node failures) and the complexities of managing concurrency. The goal of an ideal distributed system is to provide maximum throughput with acceptable latency. The CAP theorem (stating that only two out of three properties can be simultaneously satisfied) is an acronym used for reasoning about the fundamental trade-offs in the design of Distributed systems.

- C: Consistency (every read receives the most recent write or an error)
- A: Availability (every request receives a (non-error) response, without guarantee that it contains the most recent write)
- P: Partition tolerance (the system continues to operate despite arbitrary message loss (partitions))

Distributed Systems typically aim to be either AP (prioritizing Availability and Partition tolerance) or CP (prioritizing Consistency and Partition tolerance). Centralized systems are inherently AC (Availability and Consistency) because they do not inherently tolerate network partitions. Peter Deutsch formulated a list of fallacies of Distributed computing, highlighting common incorrect assumptions made by developers:

- The network is reliable (networks experience failures; always plan for retries and acknowledgments).
- There is zero latency (account for the non-negligible transmission time for packets across the network).
- Bandwidth is infinite (network bandwidth is finite; therefore, segment packet sizes appropriately).
- The network is secure (consider the security of your network, whether it is a virtual private network or a private network).
- Network topology is fixed.
- There is one administrator (distributed systems often require self-healing mechanisms as a single administrator might not be feasible).
- Transport cost is zero (similar to ”no latency,” always anticipate delays due to network transport).

- The network is homogeneous (expect that network partitions may occur).

When developing Distributed Software, it is crucial to avoid these oversimplified assumptions.

Consensus is achieved when every node agrees on the same value. Furthermore, the decided value must be one of the values that was initially proposed. Consensus is equivalent to atomic commits (atomic broadcast) [4].

Reasoning about Distributed computation can be inherently complex, necessitating the use of simplifying models. One such model involves specifying the failure assumptions, which can encompass various modes of process failure [4]:

- Crash-stop: A process halts and stops sending or receiving messages.
- Omission failures: A process fails to send or receive messages.
- Crash-recovery: A process fails but subsequently restarts, potentially recovering its state from a backup.
- Byzantine or arbitrary failures: A process exhibits unspecified or even malicious behavior, sending arbitrary messages or entering arbitrary states.
- Correct processes: Processes function as intended, reliably sending and receiving messages.

In our work, we have adopted the crash-stop model due to its relative simplicity. To achieve fault tolerance, Distributed systems must be designed to cope with failures. This is typically accomplished through a combination of techniques, including retrying lost packets, replicating data across nodes for enhanced availability, and employing mechanisms for replacing failed components (e.g., electing a new leader if the current one fails). Fault tolerance is the system's ability to maintain operation in the presence of faults.

One common use case for Distributed systems is Sharding, as depicted in Figure 3.2. In the context of databases, sharding is the process of horizontally partitioning a database table into multiple independent subsets. Sharding (also known as partitioning) is often combined with replication (duplicating data across multiple nodes) to enhance data availability. These techniques involve organizing data into chunks that may or may not overlap.

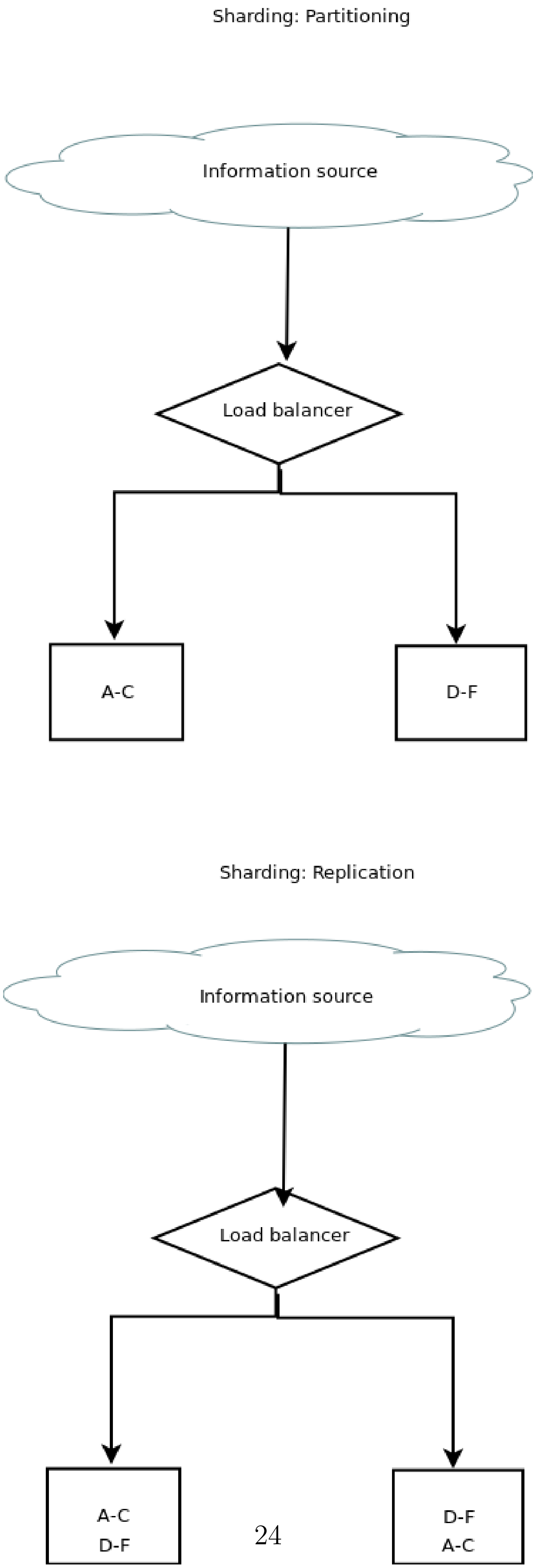


Figure 3.2: Depiction of Sharding and Replication.

NoSQL database systems tend to be more readily adaptable to distributed concepts, such as key-value stores and document stores [4, 12]. Several database architectures can ensure data availability across nodes through replication strategies (e.g., master-slave, master-master, buddy replication).

A replicated state machine is a technique used to guarantee availability in the face of failures by mirroring data and state across multiple nodes. It adheres to the following fundamental properties [4]:

- Every replica executes every operation.
- The system starts in the same state across all replicas, and if all operations are deterministic, it ends in the same final state across all correct replicas.

Nodes within a Distributed system can assume various roles [37]:

- Coordination: Managing other nodes, such as the leader in the Raft consensus algorithm.
- Cooperation: Participating in the execution of tasks to achieve a common goal.
- Dissemination: Replicating data to other nodes in the system.
- Consensus: Verifying quorums and deciding on proposed values.

A thorough understanding of network and parallel programming is a prerequisite for comprehending the intricacies of Distributed systems. We will provide a foundational overview of both these topics in Chapter 2.

3.1 Theoretical Foundations

We will discuss some fundamental theoretical concepts underpinning Distributed systems, including:

- FLP Impossibility of Consensus
- Two Generals Problem

3.1.1 FLP Impossibility of Consensus

The Fischer-Lynch-Paterson (FLP) impossibility result describes the inherent limitations of achieving consensus in asynchronous distributed systems. It states that in an asynchronous system subject to even a single crash-stop failure, it is impossible to guarantee that a consensus algorithm will always satisfy the following three crucial properties [4, 12, 37]:

- Agreement: Every correct node must eventually decide on the same value.
- Validity: If all initially proposed values are the same, then any decided value must be that value. (A weaker form states that the decided value must be one of the initially proposed values.)
- Termination: Every correct node must eventually reach a decision.

The FLP result highlights the fundamental challenges of achieving reliable consensus in systems with unpredictable timing, applicable to asynchronous, synchronous, and partially synchronous systems [4, 12, 37]:

- Consensus cannot be solved deterministically in a purely asynchronous system if there is even a single crash-stop failure.
- In a synchronous system, consensus cannot be solved if $N - 1$ nodes fail (where N is the total number of nodes).
- Consensus is solvable in a synchronous system with up to t crash failures, provided that $2t < N$ (i.e., a majority of nodes are correct).

3.1.2 Two Generals Problem

The Two Generals Problem, also known as the Coordinated Attack Problem, illustrates the difficulties of achieving agreement over an unreliable communication channel. Two generals must agree on a time to launch a joint attack, communicating only through messengers who might be captured or delayed [4, 12, 37]. Mapping this problem to a real-world Distributed system reveals the following analogous challenges [4, 12, 37]:

- Two nodes need to agree on a specific value before a defined time limit.
- Communication occurs via message passing over a potentially unreliable network channel.
- Every message ideally requires an acknowledgment to confirm receipt.

The Two Generals Problem demonstrates that it is impossible to guarantee that two parties can reach a perfect consensus in an asynchronous system when there is even a possibility of a single faulty communication link [4, 12, 37].

3.2 Logical Clocks

Time, in its ideal form, is a monotonically increasing counter with consistent intervals between increments. The duration of this interval defines the scalar units (e.g., seconds, minutes). Time becomes truly useful when it enables the ordering of events, determining precedence or simultaneity. For this ordering to be meaningful, a common reference point for initializing the counter is necessary. If clocks across different nodes are synchronized with this reference, we can accurately determine "happens-before" or concurrent relationships between events.

However, the notion of a perfectly synchronized global clock is rarely achievable in distributed systems. Therefore, it becomes crucial to determine if an event on one node was causally triggered by a previously received event from another node. This is particularly challenging in asynchronous systems, where the concept of precise timing is blurred by variable message delays across different systems. Even if individual systems possess internal clocks, synchronizing these clocks and mitigating time drift over extended periods remains a significant hurdle.

The message-passing paradigm can maintain a sense of event order by employing logical clocks, which are based on "happens-before" relationships. Logical clocks assign monotonically increasing timestamps to events to track these causal dependencies, aiming to establish a partial or total order of events in the absence of a global physical clock.

If event 'a' happens-before event 'b' on the same process, we denote this as $a \to b$, where 'a' could be a send event and 'b' a subsequent deliver event. This "happens-before" relation is fundamental for inferring event order in the absence of a global clock [4]. Two primary types of logical clocks are commonly used:

- Lamport clock
- Vector clock

To show the differences between a Lamport clock and a vector clock. We describe a distributed system comprising three processes: P1, P2, and P3. A Vector Clock is a logical time mechanism where each process maintains a vector of timestamps. The i-th element of the vector in process P_j represents P_j's knowledge of the number of events that have occurred in process P_i. This allows for the determination of causal relationships between events.

3.2.1 Lamport clock

The Lamport clock utilizes the "happens-before" relation to establish a total order of events. Timestamps generated by a Lamport clock have a defined less-than relationship (<), allowing for the ordering of any two events. However, because the timestamp is a single integer, it does not capture information about non-causality (concurrency).

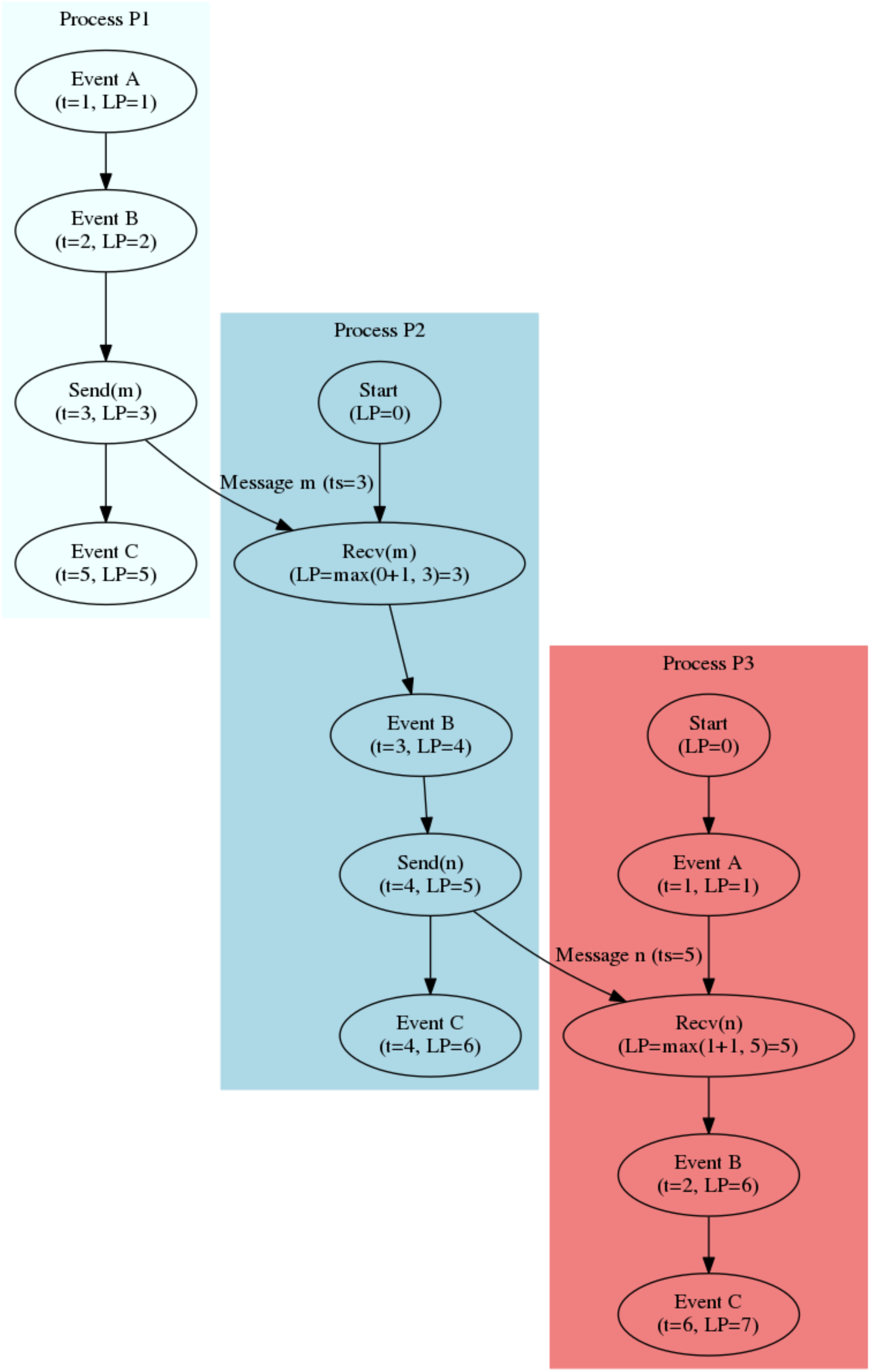


Figure 3.3: Lamport Clock Description.

The description of an example of a Lamport clock in use is shown in Figure 3.3.

Process P1

Process P1 executes a sequence of events, with its Lamport clock evolving as follows:

- **Event A (LP=1)**: The initial event in P1 is assigned a Lamport timestamp of 1.
- **Event B (LP=2)**: The subsequent internal event increments the Lamport clock to 2.
- **Send(m) (LP=3)**: Before sending message m to Process P2, the Lamport clock is incremented to 3. The timestamp of the send event is 3.
- **Event C (LP=5)**: After sending the message, an internal event C occurs, further incrementing the Lamport clock to 5.

Process P2

Process P2 starts, receives a message, executes internal events, and sends a message:

- **Start (LP=0)**: Process P2 begins with a Lamport clock value of 0.
- **Recv(m) (LP=**$\max(0+1,3)=3$**)**: Upon receiving message m with a timestamp of 3 from P1, Process P2 updates its Lamport clock to the maximum of its local clock incremented by one (1) and the received timestamp (3), resulting in a timestamp of 3 for the received event.
- **Event B (LP=4)**: An internal event B occurs, incrementing the Lamport clock to 4.
- **Send(n) (LP=5)**: Before sending message n to Process P3, the Lamport clock is incremented to 5. The timestamp of the send event is 5.
- **Event C (LP=6)**: An internal event C occurs, incrementing the Lamport clock to 6.

Process P3

Process P3 starts, receives a message, and executes internal events:

- **Start (LP=0)**: Process P3 begins with a Lamport clock value of 0.
- **Event A (LP=1)**: An initial internal event A occurs, setting the Lamport clock to 1.
- **Recv(n) (LP=**$\max(1{+}1, 5) = 5$**)**: Upon receiving message n with a timestamp of 5 from P2, Process P3 updates its Lamport clock to the maximum of its local clock incremented by one (2) and the received timestamp (5), resulting in a timestamp of 5 for the received event.
- **Event B (LP=6)**: An internal event B occurs, incrementing the Lamport clock to 6.
- **Event C (LP=7)**: A final internal event C occurs, incrementing the Lamport clock to 7.

Communication

The interaction between processes through message passing is crucial for the Lamport clock's mechanism:

- Process P1 sends message m with a timestamp of 3 to Process P2, which influences the logical time of the receive event in P2.
- Process P2 sends message n with a timestamp of 5 to Process P3, which influences the logical time of the receive event in P3.

In summary, the Lamport clock in this code snippet assigns logical timestamps to events in each process based on local increments and the timestamps of received messages, ensuring a partial ordering of events consistent with the causal "happens-before" relationship.

3.2.2 Vector clock

The Vector clock, in contrast, is based on partial order and effectively captures both causality and non-causality [13]. The ability to detect non-causality is crucial for identifying events that occur concurrently.

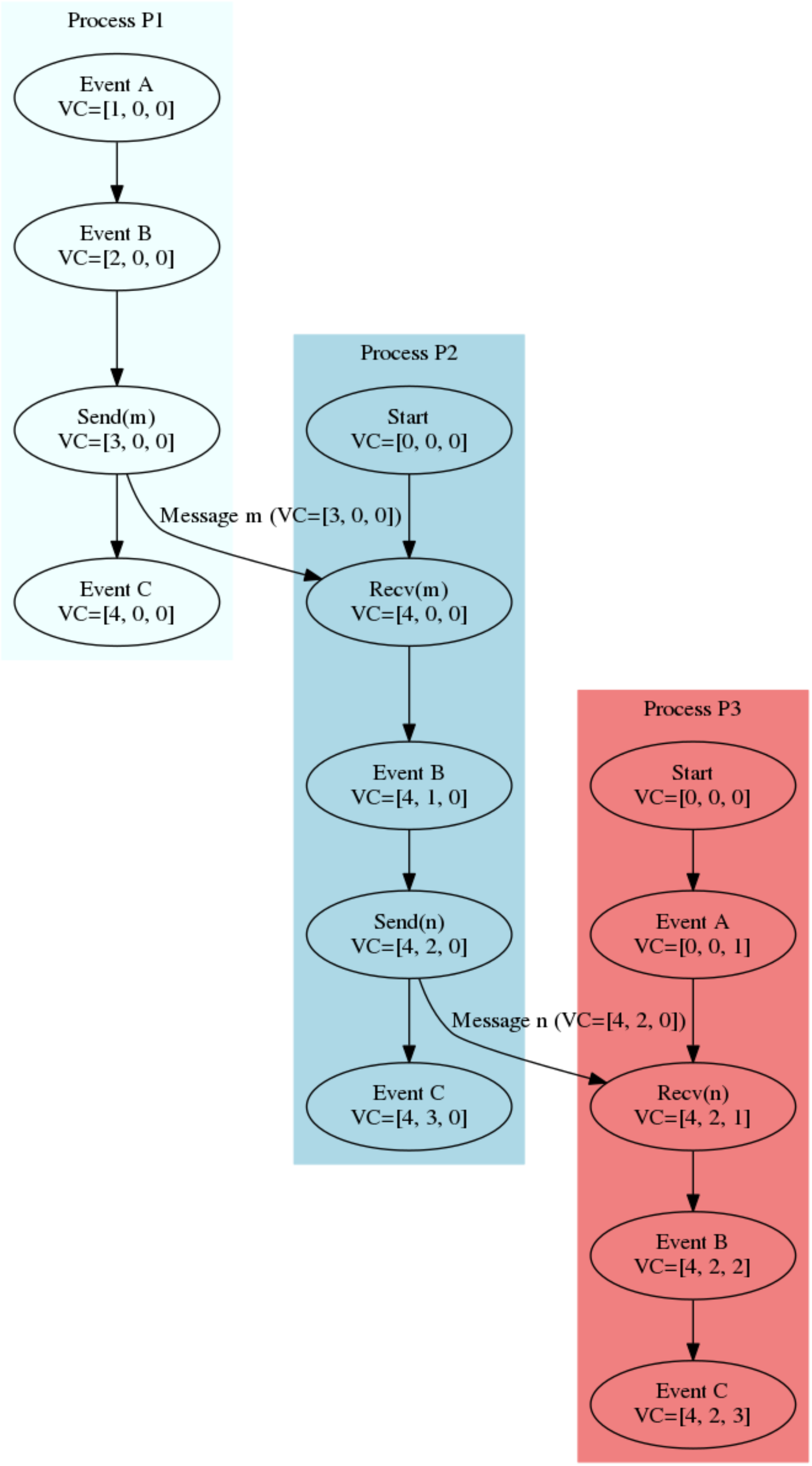


Figure 3.4: Vector Clock Description.

The description of an example of a Vector clock in use is shown in Figure 3.4.

Process P1

The evolution of the Vector Clock in Process P1 is as follows:

- **Event A (VC=[1, 0, 0])**: Upon the occurrence of the first event in P1, the first element of its vector clock (representing its event count) is incremented to 1.
- **Event B (VC=[2, 0, 0])**: The second event in P1 increments its local event count in the vector clock to 2.
- **Send(m) (VC=[3, 0, 0])**: Before sending message *m* to Process P2, P1's local event count is incremented to 3. The message *m* is sent carrying this vector timestamp.
- **Event C (VC=[4, 0, 0])**: A subsequent internal event C in P1 increments its local event count to 4.

Process P2

The Vector Clock in Process P2 evolves as it receives a message and executes local events:

- **Start (VC=[0, 0, 0])**: Process P2 initializes its vector clock with all components set to 0.
- **Recv(m) (VC=[4, 0, 0])**: When Process P2 receives message *m* with the vector timestamp [3, 0, 0] from P1, it updates its vector clock by taking the element-wise maximum of its current vector [0, 0, 0] and the received vector [3, 0, 0], and then increments its local event count (the second element), resulting in [max(0, 3)+1, max(0, 0), max(0, 0)] = [4, 0, 0].
- **Event B (VC=[4, 1, 0])**: A local event B in P2 increments its local event count to 1.

- **Send(n) (VC=[4, 2, 0])**: Before sending message n to Process P3, P2's local event count is incremented to 2. The message n is sent with this vector timestamp.

- **Event C (VC=[4, 3, 0])**: A subsequent internal event C in P2 increments its local event count to 3.

Process P3

The Vector Clock in Process P3 evolves upon receiving a message and executing local events:

- **Start (VC=[0, 0, 0])**: Process P3 initializes its vector clock with all components set to 0.

- **Event A (VC=[0, 0, 1])**: A local event A in P3 increments its local event count (the third element) to 1.

- **Recv(n) (VC=[4, 2, 1])**: When Process P3 receives message n with the vector timestamp [4, 2, 0] from P2, it updates its vector clock by taking the element-wise maximum of its current vector [0, 0, 1] and the received vector [4, 2, 0], and then increments its local event count, resulting in [max(0, 4), max(0, 2), max(1, 0)+1] = [4, 2, 1].

- **Event B (VC=[4, 2, 2])**: A local event B in P3 increments its local event count to 2.

- **Event C (VC=[4, 2, 3])**: A subsequent internal event C in P3 increments its local event count to 3.

Communication

The message exchange demonstrates the propagation of vector timestamps:

- Message m is sent from P1 with the vector timestamp [3, 0, 0] to P2, which influences P2's vector clock upon reception.

- Message n is sent from P2 with the vector timestamp [4, 2, 0] to P3, which influences P3's vector clock upon reception.

In summary, the Vector Clock mechanism illustrated in the code assigns a vector of timestamps to each event, allowing for the determination of causal relationships between events across the distributed system. The vector timestamps are updated based on local event occurrences and the vector timestamps received with messages, ensuring that each process maintains a view of the progress of all other processes that is consistent with causality.

A Vector clock can accurately determine whether two operations are concurrent or causally dependent on each other, a distinction that the Lamport clock cannot reliably make.

3.3 Failure Detector

A failure detector is a mechanism designed to identify faults (specifically, process crashes) in asynchronous distributed systems. The introduction of failure detectors leads to a refined classification of asynchronous systems known as ”Timed Asynchronous systems” or ”Partially Synchronous systems.” These systems exhibit the following characteristics [4]:

- There is no guaranteed upper bound on message delivery time.
- There is no guaranteed upper bound on process computation time.
- However, the drift rate of local clocks is known and bounded.

A typical failure detection algorithm operates as follows [4, 12, 37]:

- Each node in the system employs a local failure detector.
- Initially, the failure detector's suspicions might be incorrect, but it is designed to become eventually accurate.
- Nodes periodically exchange heartbeat messages with all other processes they believe to be alive.
- If a node does not receive a heartbeat response from another process within a predefined timeout period, it begins to suspect that process.

- If a message is subsequently received from a suspected node, the suspicion is typically revised (the node is no longer suspected), and the timeout period for that node might be increased.

- Otherwise, if no communication is received after repeated timeouts, the failure detector concludes that the process has crashed.

In asynchronous systems, consensus and atomic broadcast become solvable with the aid of a failure detector. For a failure detector to be practically useful, it must satisfy certain requirements with varying degrees of certainty [4, 12, 37]:

- Completeness: (How promptly are crashed nodes detected?)
  Every process that has crashed is eventually detected by every correct (non-crashed) process (this relates to liveness).

- Accuracy: (How often are alive nodes mistakenly suspected?)
  No correct process is ever suspected by any other correct process (this relates to safety).

These requirements can be further categorized as strong or weak. In our implementations, we prioritized completeness over strong accuracy. In our specific use case, it is acceptable for a healthy process to be temporarily suspected, especially during intermittent network issues, as its status can be updated back to "alive" upon subsequent communication. We initially assumed all processes were healthy. Depending on the specific application requirements, it might be preferable to initially assume all processes are dead and require explicit confirmation of their liveness. When considering adding a timeout outside a busy-wait loop for reads, the timeout duration needs to be estimated relative to the arrival of the first message. If the arrival of the first message is not guaranteed (e.g., due to a lack of initial quorum), then the system might not function correctly from the outset.

To design a failure detector optimized for accuracy, measures are taken to minimize the suspicion of healthy processes. A common strategy is to incrementally increase the timeout period if a node fails to respond to pings, rather than immediately suspecting it as failed. This "benefit of the doubt" approach acknowledges that network communication issues might be the cause of the lack of response, rather than a process crash. By maintaining per-node timeout values and dynamically adjusting them based on communication history, the failure detector can learn the typical behavior of the network and become more resilient to transient network faults. Failure detectors can be further enhanced by expanding the types of failures they can identify. Our implementation targets a minimal subset of potential failures. Ideally, the design of a failure detector should leverage domain-specific knowledge about the types of failures that are most likely to occur in a given system, allowing for tailored detection mechanisms.

3.4 Graceful Degradation

Distributed systems have numerous industrial applications as the Internet becomes increasingly ubiquitous. Organizations are building these large-scale services with stringent requirements for minimal downtime. Given the complexities of these interacting nodes, a myriad of issues can arise, including node failures and intermittent network calls, among others.

Graceful degradation (e.g., using a circuit breaker pattern) is a situation where, as more nodes fail, the system's functionality is reduced in a controlled manner rather than experiencing a catastrophic breakdown. This can involve reducing the components in the user interface, changing functionality as network bandwidth is throttled, or ensuring that only the most critical services remain running as the number of nodes in the cluster decreases to a bare minimum. In contrast, a quorum-based system (e.g., using Paxos or Raft) represents a form of achieving bounded graceful degradation as a fault-tolerance mechanism. These systems stop working when the number of operational nodes falls below the quorum limit.

Failure tolerance is at the core of building fault-handling mechanisms using resiliency patterns (e.g., retry, circuit breaker, fail-fast, bulkhead) [11]. The overarching aim of these patterns is to prevent requests from being directed to faulty resources, thereby enhancing the resilience of the distributed systems.

- A circuit breaker makes use of a failure detector that scans for breakdowns in your cluster. A failure is determined based on a set threshold (consider using the accuracy and completeness properties). Once a failure rate threshold is exceeded, the resource becomes unavailable and hence cannot accept new requests until a set time has elapsed, at which point it is assumed that the unavailable resource has recovered and can begin to process requests again. The recovery process is then re-triggered by a secondary resource, e.g., placing the circuit breaker behind a load balancer. When you visit a website and see a banner indicating that the site is under maintenance, you have probably been redirected by a circuit breaker in action.

- The retry pattern would suffice for transient failures, which are short-lived errors. However, for prolonged failures, the circuit breaker is ideal. Care must be taken when using the circuit breaker to handle cascading failures, which occur when downstream systems become non-functional due to dependency on an unavailable parent system. Correlation vectors can help with distributed debugging of such failures.

It is important to consider the following recommendations when using resiliency patterns that utilize timeouts. Timeouts can hold up resources when limits are reached. Unfortunately, the risk of cascading failures on requests that are dependent on other requests can lead to a catastrophic breakdown of the system. Setting appropriate time limits for timeouts must be done in a way that delayed processes are not timed out prematurely.

Chapter 4

Distributed Consensus Algorithms

"Beware the man of a single book." – Thomas Aquinas

Achieving agreement among nodes is a fundamental objective that Distributed systems strive for as a core component of performing real-world tasks. Abortable consensus offers a mechanism for reaching consensus through repeated attempts across multiple rounds. The overall consensus process concludes when convergence is achieved, which occurs when a majority or all of the nodes have decided on the same value. We will discuss several well-known distributed consensus algorithms, as detailed in Sections 4.1, 4.2, and 4.3, respectively.

4.1 Paxos Algorithm

Paxos is a fundamental algorithm for achieving consensus among multiple nodes across several rounds of communication. It provides a method for achieving abortable consensus in a distributed system with the following key properties [12]:

- Single point of failure.
- Accommodation of network partitions.

The algorithm generally follows these steps [4, 37]:

- Each node in the system is assumed to have a network channel enabling communication with every other node.
- Only designated proposers can suggest values to be decided by the consensus algorithm.

- Upon reaching consensus, every node in the system will have decided on the same value.

The following concurrency-related properties [4, 12, 37] are generally applicable in parallel and concurrent programming paradigms:

- Liveness: A majority of the nodes must be able to reliably communicate to exchange values and make progress.
- Safety: Only values proposed by proposers can be chosen as the consensus value.

The main characteristics of the Paxos algorithm are [4, 37, 12]:

- Abortable consensus: Multiple rounds of communication might be necessary before a value is agreed upon.
- Agreement based on a quorum: A quorum (typically a majority) of nodes must decide on a value for it to be considered the consensus value.

The Paxos algorithm defines several roles that nodes in the system can assume:

- Client: The entity that initiates the consensus process by proposing a value.
- Proposer: A node that receives a proposal from a client and attempts to get it accepted by the acceptors.
- Acceptor: A node that receives proposals and votes from proposers, deciding whether to accept a proposed value.
- Learner: A node that learns the decided value once a quorum of acceptors has accepted it.

We will now discuss Single Value Paxos and Sequence Paxos in the subsequent subsections.

Single Valued Paxos

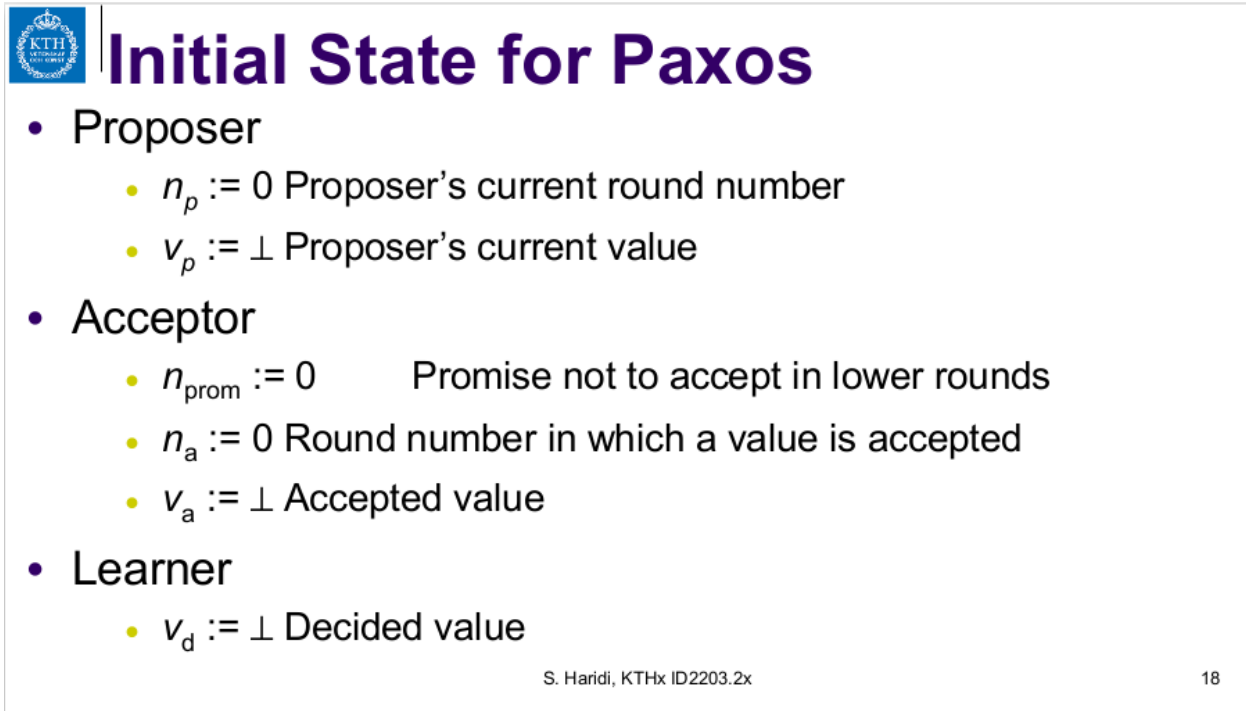
The Single Valued Paxos algorithm proceeds as follows [4, 12]:

- A client sends a proposed value to one or more proposers.
- Proposers send prepare requests (containing a proposal number) to a quorum of acceptors.
- Proposers receive responses from the acceptors. If a quorum of acceptors responds, the proposer proceeds to the next phase.
- Proposers send accept requests (containing the proposal number and a proposed value) to the same quorum of acceptors. The value might be the original proposed value or the highest-numbered value already accepted by any of the acceptors in their responses to the prepare request.
- Once a quorum of acceptors has accepted an accept request for a particular value, that value is considered chosen.
- The chosen value is then communicated to the set of learners.

Properties of the Paxos algorithm [4, 12]:

- Validity: If a value is decided, it must have been proposed by some client.
- Uniform agreement: No two processes decide on different values.
- Integrity: Each process can decide on at most one value.
- Termination: Every correct process eventually decides on a value, assuming a majority of processes are correct and reliable communication exists.

The implementation of the Single Value Paxos algorithm discussed in this book is visually represented in Figure 4.1 and Figure 4.2. Figure 4.1 illustrates the initial states of each node at the internal start of the Single Value Paxos process.



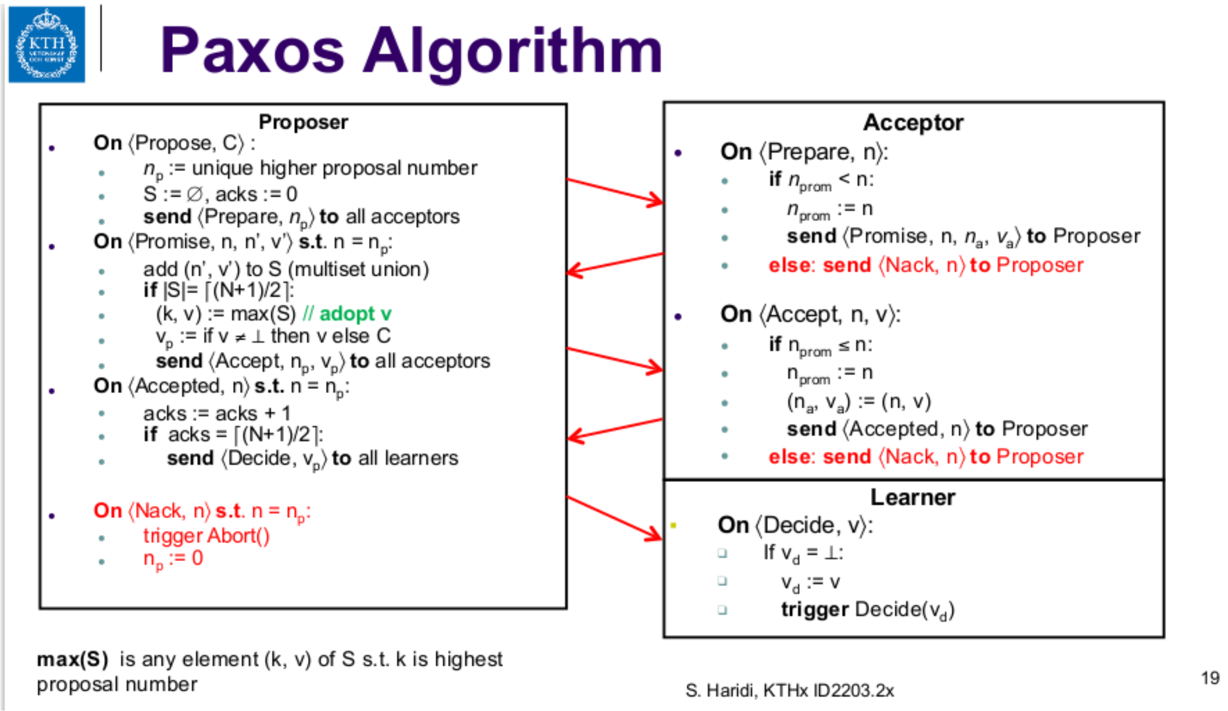
Initial State for Paxos

- Proposer
  - n_p := 0 Proposer's current round number
  - v_p := ⊥ Proposer's current value
- Acceptor
  - n_{prom} := 0 Promise not to accept in lower rounds
  - n_{a} := 0 Round number in which a value is accepted
  - v_{a} := ⊥ Accepted value
- Learner
  - v_{d} := ⊥ Decided value

S. Haridi, KTHx ID2203.2x 18

Figure 4.1: Initialization of states at the internal start of Paxos Algorithm [4].

Figure 4.2 presents a sequence (interaction) diagram depicting the messages exchanged between the different roles (Client, Proposer, Acceptor, Learner) in the network during the Single Value Paxos algorithm.



Paxos Algorithm

Proposer

```
On ⟨Propose, C⟩ :
    $n_p$ := unique higher proposal number
    S := ∅, acks := 0
    send ⟨Prepare, $n_p$⟩ to all acceptors
On ⟨Promise, n, n', v'⟩ s.t. n = $n_p$:
    add (n', v') to S (multiset union)
    if |S|= ⌈(N+1)/2⌉:
        (k, v) := max(S) // adopt v
        $v_p$ := if v ≠ ⊥ then v else C
        send ⟨Accept, $n_p$, $v_p$⟩ to all acceptors
On ⟨Accepted, n⟩ s.t. n = $n_p$:
    acks := acks + 1
    if acks = ⌈(N+1)/2⌉:
        send ⟨Decide, $v_p$⟩ to all learners

On ⟨Nack, n⟩ s.t. n = $n_p$:
    trigger Abort()
    $n_p$ := 0
```

Acceptor

```
On ⟨Prepare, n⟩:
    if $n_{prom}$ < n:
        $n_{prom}$ := n
        send ⟨Promise, n, $n_a$, $v_a$⟩ to Proposer
    else: send ⟨Nack, n⟩ to Proposer

On ⟨Accept, n, v⟩:
    if $n_{prom}$ ≤ n:
        $n_{prom}$ := n
        ($n_a$, $v_a$) := (n, v)
        send ⟨Accepted, n⟩ to Proposer
    else: send ⟨Nack, n⟩ to Proposer
```

Learner

```
On ⟨Decide, v⟩:
    If $v_d$ = ⊥:
        $v_d$ := v
        trigger Decide($v_d$)
```

max(S) is any element (k, v) of S s.t. k is highest proposal number

S. Haridi, KTHx ID2203.2x

19

Figure 4.2: Sequence of message flow in a Paxos Algorithm [4].

It's worth noting that there are several variations of Single Value Paxos, including Fast Paxos, Egalitarian Paxos, and Flexible Paxos [37], each offering different performance characteristics or fault-tolerance trade-offs.

Sequence Paxos

The Sequence Paxos algorithm extends the Single Value Paxos to achieve agreement on a sequence of values (often representing a log of events) in a consistent order across all nodes [4, 37, 12]. The general approach involves:

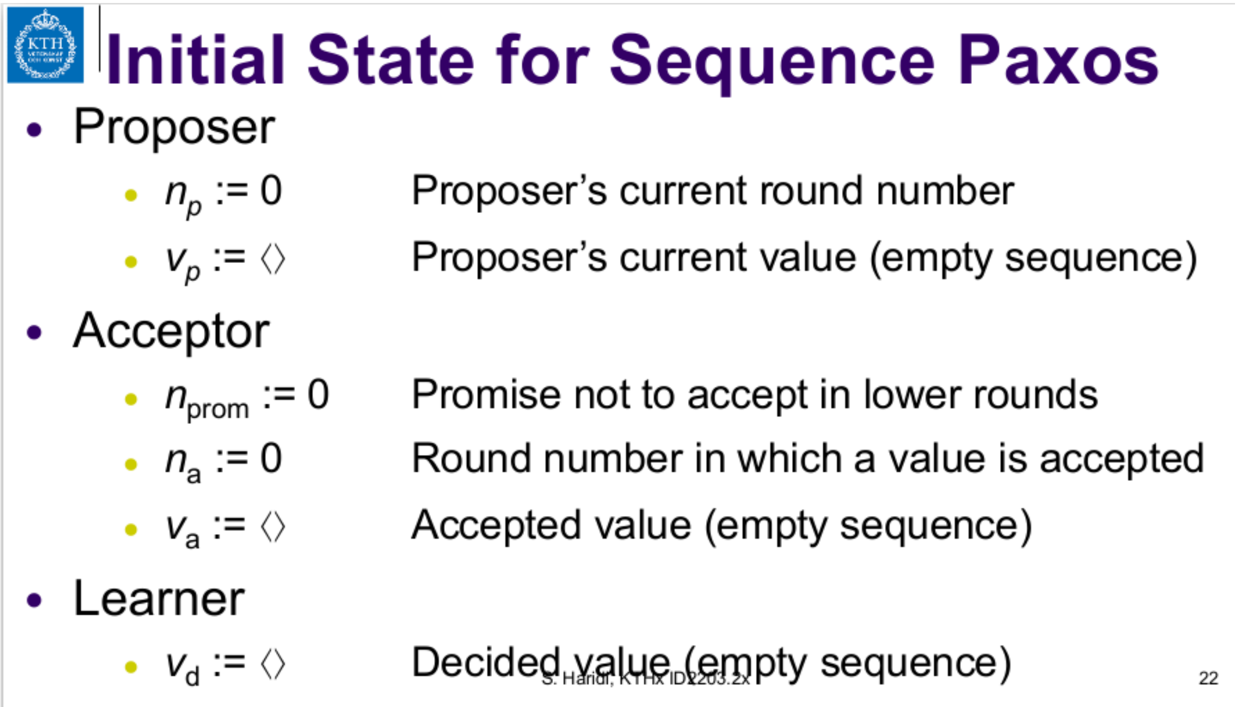
- Repetitively performing the Single Value Paxos algorithm for each position (slot) in the sequence, ensuring that the decisions for each slot are made in order.
- Maintaining a strict prefix invariant: If all nodes have decided on the values for the first i slots, then these decided prefixes must be identical across all nodes.
- Effectively implementing an ordered atomic broadcast mechanism, where all messages are delivered to all correct processes in the same order.

In more detail, the fundamental principles of the Sequence Paxos algorithm include:

- Achieving agreement on a common ordering of events or operations across all participating processes.
- Constructing a consistent log of decided values by running multiple instances of the Single Value Paxos algorithm, one for each position in the log.

The implementation of the Sequence Paxos algorithm discussed in this book is illustrated in Figure 4.3, Figure 4.4, and Figure 4.5.

Figure 4.3 shows the initial internal states of each node at the beginning of the Sequence Paxos process.



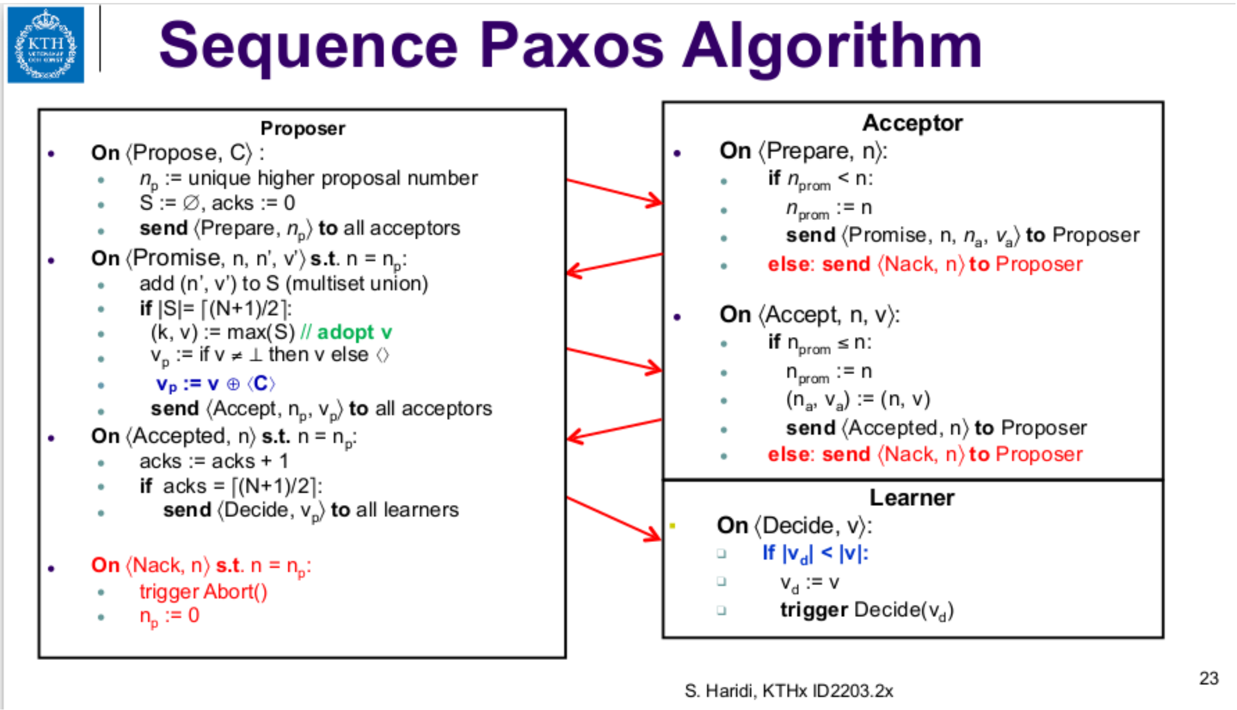
Initial State for Sequence Paxos

- Proposer
  - $n_p := 0$ Proposer's current round number
  - $v_p := \langle\rangle$ Proposer's current value (empty sequence)
- Acceptor
  - $n_{\mathrm{prom}} := 0$ Promise not to accept in lower rounds
  - $n_{\mathrm{a}} := 0$ Round number in which a value is accepted
  - $v_{\mathrm{a}} := \langle\rangle$ Accepted value (empty sequence)
- Learner
  - $v_{\mathrm{d}} := \langle\rangle$ Decided value (empty sequence)

S. Haridi, KTHx ID2203.2x 22

Figure 4.3: Initialization of internal states at the start of Sequence Paxos Algorithm [4].

Figure 4.4 presents a sequence (interaction) diagram illustrating the message flow between the different roles in the network during the Sequence Paxos algorithm.



Sequence Paxos Algorithm

```
Proposer
On ⟨Propose, C⟩ :
    $n_p$ := unique higher proposal number
    S := ∅, acks := 0
    send ⟨Prepare, $n_p$⟩ to all acceptors
On ⟨Promise, n, n', v'⟩ s.t. n = $n_p$:
    add (n', v') to S (multiset union)
    if |S|= ⌈(N+1)/2⌉:
        (k, v) := max(S) // adopt v
        $v_p$ := if v ≠ ⊥ then v else ⟨⟩
        $v_p$ := v ⊕ ⟨C⟩
        send ⟨Accept, $n_p$, $v_p$⟩ to all acceptors
On ⟨Accepted, n⟩ s.t. n = $n_p$:
    acks := acks + 1
    if acks = ⌈(N+1)/2⌉:
        send ⟨Decide, $v_p$⟩ to all learners

On ⟨Nack, n⟩ s.t. n = $n_p$:
    trigger Abort()
    $n_p$ := 0
```

```
Acceptor
On ⟨Prepare, n⟩:
    if $n_{prom}$ < n:
        $n_{prom}$ := n
        send ⟨Promise, n, $n_a$, $v_a$⟩ to Proposer
    else: send ⟨Nack, n⟩ to Proposer

On ⟨Accept, n, v⟩:
    if $n_{prom}$ ≤ n:
        $n_{prom}$ := n
        ($n_a$, $v_a$) := (n, v)
        send ⟨Accepted, n⟩ to Proposer
    else: send ⟨Nack, n⟩ to Proposer
```

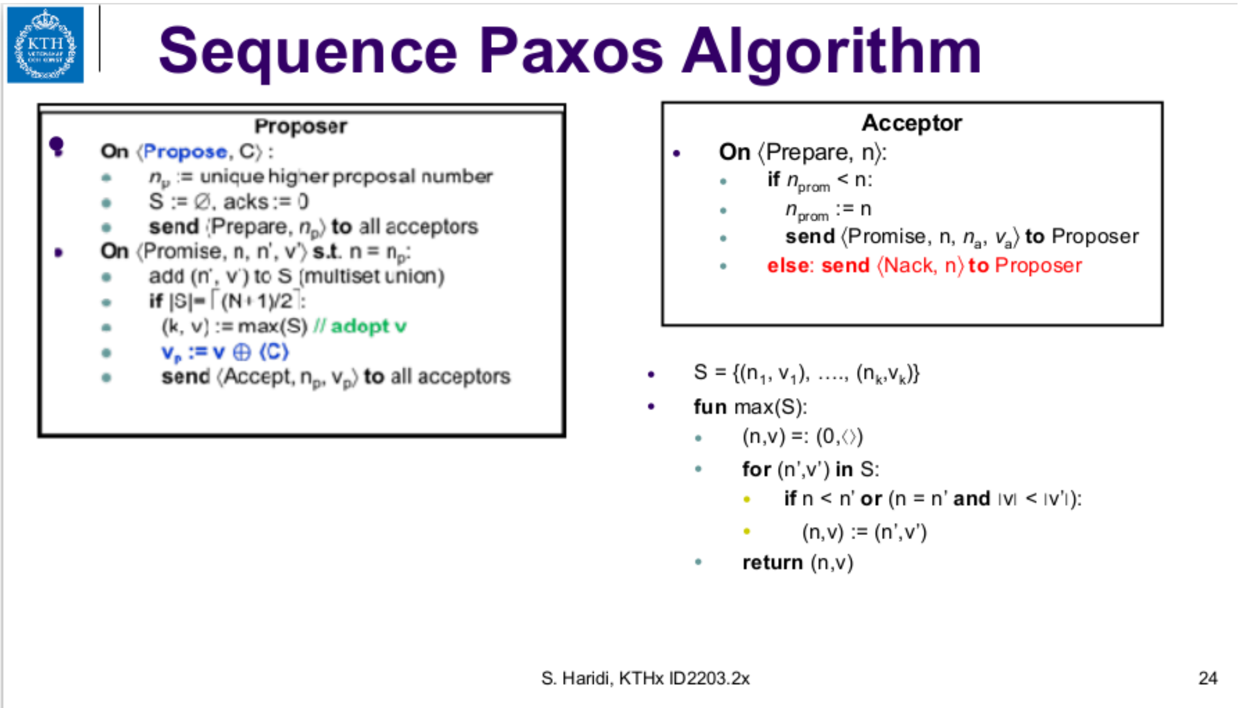
```
Learner
On ⟨Decide, v⟩:
    If |$v_d$| < |v|:
        $v_d$ := v
        trigger Decide($v_d$)
```

S. Haridi, KTHx ID2203.2x

23

Figure 4.4: Message flow in a Sequence Paxos Algorithm [4].

Figure 4.5 highlights the algorithmic adjustments and mechanisms employed to maintain the crucial prefix invariants in the log as the Sequence Paxos algorithm progresses.



Sequence Paxos Algorithm

```
Proposer
On ⟨Propose, C⟩ :
    $n_p$ := unique higher proposal number
    S := ∅, acks := 0
    send ⟨Prepare, $n_p$⟩ to all acceptors
On ⟨Promise, n, n', v'⟩ s.t. n = $n_p$:
    add (n', v') to S (multiset union)
    if |S|= ⌈(N+1)/2⌉:
      (k, v) := max(S) // adopt v
      $v_p$ := v ⊕ ⟨C⟩
      send ⟨Accept, $n_p$, $v_p$⟩ to all acceptors
```

```
Acceptor
On ⟨Prepare, n⟩:
    if $n_{prom}$ < n:
      $n_{prom}$ := n
      send ⟨Promise, n, $n_a$, $v_a$⟩ to Proposer
    else: send ⟨Nack, n⟩ to Proposer
```

```
S = {($n_1$, $v_1$), …., ($n_k$,$v_k$)}
fun max(S):
    (n,v) =: (0,⟨⟩)
    for (n',v') in S:
        if n < n' or (n = n' and |v| < |v'|):
            (n,v) := (n',v')
    return (n,v)
```

S. Haridi, KTHx ID2203.2x 24

Figure 4.5: Maintaining prefix of log invariants in a Sequence Paxos Algorithm [4].

4.2 Election Algorithm

In distributed systems, allowing multiple nodes to propose values concurrently can introduce complexities in coordinating these proposals and can lead to increased contention for system resources. To streamline the consensus process and minimize such conflicts, it is often advantageous to designate a single node as the leader. This leader then assumes the role of a central coordinator, responsible for initiating and managing the agreement process. Our approach to leadership election relies on an underlying failure detector. This is a crucial component that continuously monitors the health and responsiveness of the participating nodes to identify potential failures or unresponsiveness of the currently designated leader [4, 37]. (For a more detailed explanation of failure detectors, please refer to the preceding section.)

We are using a variation of the bully algorithm which is described below [37]. The algorithm does the following:

- A server can become a candidate by waiting for some set time and if no ping

from others is received. It becomes a candidate.
Note: timeout must be longer than the duration of the leader election.
The choice of delay has a significant impact on leader election.
The rule of thumb for deciding the duration

– If the value is set too low, then the second candidate begins the election before the first election triggered by the first candidate.

– if too high, then it will take too long for the election to start after the old leader has died. The new candidate starts an election.

In summary, the algorithms can be described as follows: When a server suspects that the current leader has failed or become unresponsive (this suspicion typically arises from the absence of expected communication, such as periodic heartbeat messages, within a predefined timeout period, as determined by the failure detector), it initiates an election process. The goal of this election is to select a new leader from the set of currently active and available servers.

Here is a diagram showing how leader election works as shown in Figure 4.6.

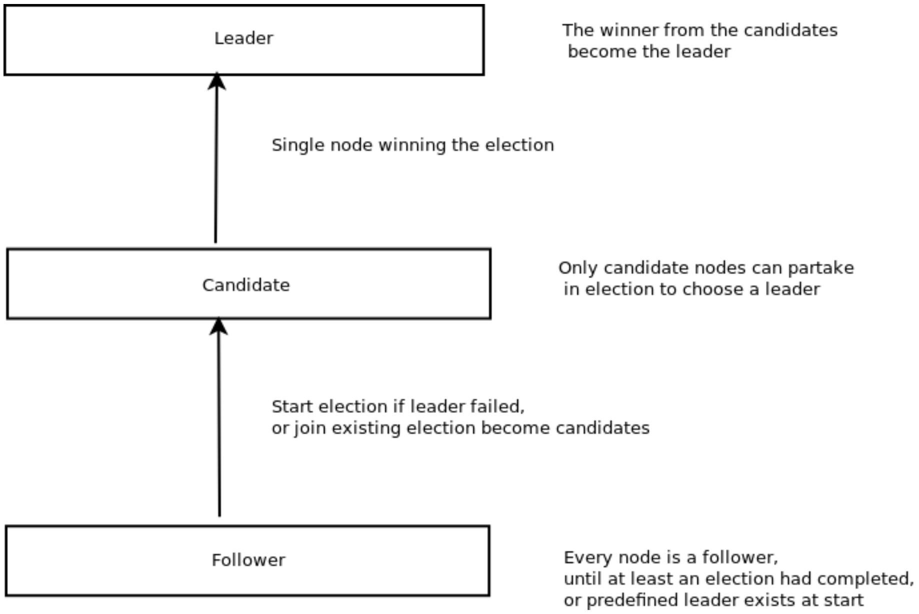

Leader
The winner from the candidates become the leader
Single node winning the election
Candidate
Only candidate nodes can partake in election to choose a leader
Start election if leader failed, or join existing election become candidates
Follower
Every node is a follower, until at least an election had completed, or predefined leader exists at start

Figure 4.6: Depiction of Leader Election.

A variation using ballots for a leadership election to choose a leader from a set of candidates is shown below. The following code snippet presents a ballot-based approach for electing a leader from a set of candidate servers within an MPI (Message Passing Interface) environment as seen in leader-election3.c:

```
// Function responsible for initiating and managing the leader election process.
void leaderElection(int my_rank, int num_procs, int *leader,
MPI_Datatype mpi_data_type) {
    process_cnts = create_shared_var(MPI_COMM_WORLD, 0);
    int cnt = 0; // Local counter for the current election attempt
    int run_loop = 1;
    // Example timeout duration for detecting leader failure
    double duration = 2.0;
    double beg = MPI_Wtime();
    double end = beg;
    int msg_flag;
    MPI_Status status;
    int is_leader_dead = (*leader == MPI_PROC_NULL);
    // Initially true if no leader is known
    MPI_Request requests[num_procs];

    while (run_loop) {
        is_leader_dead = isLeaderFailed((*leader));
        if (is_leader_dead) {
            beg = MPI_Wtime();
            while ((end - beg) <= duration) {
                end = MPI_Wtime();
                MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &msg_flag, &status);
                // Break the timeout loop if any message is received
                if (msg_flag) break;
            }
            // Check again if another process has already started an election
            // and potentially updated the leader.
            is_leader_dead = isLeaderFailed((*leader));
            if (is_leader_dead) {
                printf("Process %d: A leader process (pid=%d)
has failed or no initial leader\n", my_rank, *leader);
                data package;
                memset(&package, 0, sizeof(package));
                // Generate a unique ballot based on time
                package.ballot = (int)time(NULL);
                package.pid = my_rank;

                printf("Process %d: An election has been triggered
with ballot %d\n", my_rank, package.ballot);
                beginElection(package, num_procs, requests, mpi_data_type);
            }
        }
        // Increment the shared counter to track the number of processes
        // that have reached this point in the election cycle.
        cnt = increment_counter(process_cnts, 1);
        // Determine if a quorum of processes has participated in this round.
        if (cnt == (int) MAX(((num_procs + 1) / 2.0), 1)) {
            run_loop = 0; // Exit the election loop once a quorum is reached.
        }
    }
    if (process_cnts != NULL && process_cnts->win != MPI_WIN_NULL) {
        SAFE_MPI_CALL(MPI_Win_free(&process_cnts->win));
        free(process_cnts);
    }
}

// Function for a process to check incoming election messages and potentially
// update the current leader based on the received ballot.
void checkLeader(data recv, int my_rank, int num_procs, int *leader,
MPI_Request requests [], MPI_Datatype mpi_data_type) {
    // Static local variable to track the highest ballot seen
    static int max_ballot = -10000;
    // Static local counter for the number of promises received
    static int cnt = 0;
    enum msgTag ctag;
    data accept_value;
    memset(&accept_value, 0, sizeof(accept_value));

    // If the received ballot is higher than the current maximum, update it.
    if (recv.ballot > max_ballot) {
        max_ballot = recv.ballot;
        // Store the data associated with the highest ballot.
        accept_value = recv;
    }

    // Increment the shared counter for the number
    // of processes that have responded.
    cnt = increment_counter(proc_cnts, 1);
    int promise_cnt = (int) MAX(((num_procs + 1) / 2.0), 1); // Quorum size.

    // If a quorum of processes has responded and the current leader is not
    // the process with the highest ballot, then broadcast the new leader.
    if (cnt == promise_cnt && (*leader != accept_value.pid)) {
        ctag = mSetLeader;
        printf("Process %d: Sending broadcast to set leader to %d
with ballot %d\n", my_rank, accept_value.pid, accept_value.ballot);
        *leader = accept_value.pid; // Update the local leader.
        for (int other_rank = 0; other_rank < num_procs; other_rank++) {
            SAFE_MPI_CALL(MPI_Isend(&accept_value, 1,
mpi_data_type, other_rank, ctag, MPI_COMM_WORLD, &requests[other_rank]));
        }
    }
}
```

Note

Remember to reset both counters for both functions at the end of the leader election.

It is crucial to remember to reset the shared counters (‘process_cnts‘, ‘proc_cnts‘, and ‘process_max_ballot‘) and to release their associated MPI Window objects using ‘MPI_Win_free()‘ at the end of the leader election process. Failure to do so can lead to interference with subsequent MPI operations. The provided ‘resetElectionCounters()‘ function attempts to perform this cleanup. Additionally, you will need to free the MPI datatype ‘mpi_data_type‘ using ‘MPI_Type_free()‘.

The ‘MPI_Barrier(MPI_COMM_WORLD)‘ calls are essential for ensuring that all processes synchronize at specific points within the algorithm. In the ‘leaderElection()‘ function, the barrier ensures that all processes have completed their initial leader failure detection and potential election triggering before proceeding to the next iteration of the loop. In the ‘main()‘ function, barriers are used to synchronize the printing of leader information and to ensure that all processes participate in the ‘checkLeader()‘ phase after receiving potential leader announcements.

The provided code snippet includes placeholders and assumes the existence of atomic primitives (‘create_shared_var‘, ‘increment_counter‘, ‘reset_var‘) that would typically be implemented using MPI Windows for shared memory access and synchronization, as potentially demonstrated in your Paxos implementation. You will need to ensure that these primitives are correctly implemented and accessible within your election algorithm code. The ‘isLeaderFailed()‘ function is a simplified placeholder for failure detection and should be replaced with a more robust mechanism (e.g., monitoring heartbeats) in a production-ready system. The ‘checkLeader()‘ function now uses static local variables (‘max_ballot‘ and ‘cnt‘) to maintain state across multiple invocations within a single election cycle. The ‘initializeElectionCounters()‘ function is added for the proper initialization of the shared counters and variables before the election begins. The ‘SAFE_MPI_CALL‘ macro is included as a good practice for handling potential errors in MPI function calls. The ‘main()‘ function provides a basic example of how the ‘leaderElection‘ and ‘checkLeader‘ functions might be used within an MPI program.

Add the required barriers to make the example work as expected. Look at my implementations of Paxos on how to create proper atomic primitives.

4.3 Raft Algorithm

The Paxos algorithm, while foundational for achieving consensus in distributed systems, is often perceived by software engineers as complex and challenging to reason about. This complexity motivated the invention of the Raft algorithm, designed to be a more understandable alternative while providing equivalent functionality. Raft is described as a leader election-based sequence Paxos, essentially integrating the concepts of Paxos, a replicated log, and a designated leader [4, 12]. The elected leader acts as the sole proposer, centralizing coordination and minimizing potential contention. To ensure fault tolerance, Raft employs leader election, which is triggered if the current leader is detected as dead.

The Raft algorithm defines three distinct roles for the participating nodes:

- Candidate: A node that initiates an election with the goal of becoming the leader.

- Leader: The candidate that successfully wins the election and assumes the responsibility of coordinating the consensus process.

- Follower: A participant in the Raft cluster who is not currently a candidate or a leader. Followers passively listen for instructions from the leader and vote in elections.

Despite its design for simplicity, Raft can still encounter potential issues:

- Multiple leaders: Although Raft is designed to have a single leader, scenarios such as network partitions or subtle timing issues during leader election could theoretically lead to the temporary existence of multiple nodes believing they are the leader. This can result in conflicting proposals and compromise the correctness of the consensus algorithm.

- No leader: Problems during the leader election process, such as unhandled ties in votes or persistent faults affecting potential candidates, can lead to a situation where no leader is successfully elected. Without a leader, the Raft cluster cannot make progress on proposing and committing new log entries.

- Missing log entries: Unexpected errors, such as node crashes or network disruptions, could potentially cause some decided log entries to be lost or not fully replicated across the cluster.

- Divergent logs: If the cluster experiences issues like multiple leaders (even transiently) or failures during log replication, different nodes might end up with logs containing conflicting decided values at the same index. This divergence violates the fundamental consistency property that Raft aims to maintain.

Let us consider the following questions in the context of Raft:

- Can an election choose multiple leaders [12]? Yes, although Raft's mechanisms are designed to strongly discourage this, transient network issues or timing-dependent failures could theoretically lead to split votes and the possibility of multiple nodes believing they are leaders for a short period.

- Can an election fail to choose a leader [12]? Yes, scenarios such as all candidates failing or a persistent split vote across the cluster could result in no single candidate obtaining a majority, thus leading to a failed election.

In our implementation of leader election (as discussed in the previous section), we incorporated careful delay management and election logic to significantly reduce the practical likelihood of these issues occurring. However, the theoretical possibility remains, especially in challenging network environments.

As noted, if both the answers to the questions above are true (multiple leaders or no leader), then the risk of divergent logs becomes considerably higher.

Raft shares significant similarities with both leader election mechanisms and sequence Paxos. Given that we have already implemented both of these foundational components, implementing the Raft algorithm by combining their functionalities becomes a relatively straightforward task. The following pseudocode snippets illustrate how this combination could be structured, resembling actual source code organization:

Algorithm 1: Basic role assignment based on the elected leader.

```
if (rank == leader)
{
 // This node acts as the proposer,
    // initiating and coordinating log replication.
}
else
{
    // Example: Assign nodes with even rank as acceptors.
 isAcceptor = rank % 2;
 if (isAcceptor)
 {
 // This node acts as an acceptor, voting on proposed log entries.
 }
 else
 {
 // This node acts as a learner, receiving
        // and storing committed log entries.
 }
}
```

Algorithm 2: Demonstrating an alternative grouping strategy with a single acceptor.

```
if (rank == leader)
{
 // This node acts as the proposer.
}
else
{
    // Example: The node with rank immediately
    // following the leader is the learner.
 isLearner = (leader + 1) % n;
 if (isLearner)
 {
 // This node acts as the learner.
 }
 else
 {
 // This node acts as the acceptor. In this example,
        // there is only one acceptor (excluding the leader and learner).
 }
}
```

These algorithms demonstrate the flexibility in assigning roles within a distributed consensus system once a leader has been elected. The specific grouping and role assignment strategies can be tailored based on the desired fault tolerance characteristics and performance considerations of the Raft implementation. As suggested, the possibilities are limited only by our imagination in how we structure the roles of acceptors and learners once a stable leader has been established through the leader election process.

4.4 Stabilization Algorithm

This algorithm permits the return of a distributed computation to a ’correct behavior’ from a perturbed state [33]. This perturbation may be due to failures, such as those affecting networks and nodes. Self-repairing algorithms guarantee correct behavior even in the presence of failures [45]. Robust algorithms, as discussed in Chapter 4, have a fixed bound on the number and types of failures they can tolerate while exhibiting graceful degradation. In contrast, stabilization algorithms allow for any number of failures, with an expected delay before incorrect operation may occur until the failure is repaired and the system returns to a correct state. Stabilization algorithms are ideal for transient errors, as discussed in Section 3.4. Most robust algorithms follow a crash-stop model, whereas stabilization algorithms follow a crash-recovery model. Stabilization can be achieved through several means, including eliminating illegal states (domain restriction) and snapshotting.

Eliminating illegal states can be illustrated with the example of a mobile phone’s torchlight. There is a possibility that the torchlight might be accidentally switched on, potentially draining the battery. This condition can be avoided by redesigning the interface to require the user to continuously press a button to activate the light; releasing the button would switch the light off. Thus, the illegal state of the light being indefinitely switched on is eliminated.

Snapshotting involves saving the configuration states of a distributed computation for later retrieval [43, 45]. When an error is detected, the system can revert to a previously known good configuration saved in the snapshot (a change point). Snapshotting is desirable when manual intervention to resolve errors is costly. Retrieving the saved change point when an illegal state is reached can guarantee self-healing in the distributed computing system. For reliability, it is essential that the saved change point has not been compromised by a malicious node, ensuring the system's return to correct behavior.

We have implemented a mechanism for saving the decided value in a single-valued Paxos algorithm. The following outline is a snippet from the file named single-paxos3-snapshot.c. This implementation includes some custom details that are not part of the general Paxos specification. Specifically, I created a new role, 'telemetry,' in addition to the standard proposer, acceptor, decider, and learner roles. I also created a dedicated communicator for the telemetry role to simulate logging the distributed system's state as quickly as possible. This is a simplification, as we have omitted the data storage logic.

```
enum manageTag {mSNAPSHOT, mREVERT};

void trigger_snapshot(MPI_Comm comm, data payload_backup,
 int num_procs, MPI_Datatype mpi_data_type, MPI_Request requests[]) {
    enum manageTag ctag = mSNAPSHOT;
    for (int other_rank = 0; other_rank < num_procs; other_rank++)
    {
        MPI_Isend(&payload_backup, 1, mpi_data_type, other_rank,
ctag, comm, &requests[other_rank]);
    }
}

void handle_snapshot_messages(MPI_Comm comm, int my_rank,
MPI_Request requests[], MPI_Status status[],
MPI_Datatype mpi_data_type,
struct mpi_counter_t *telemetry_msg_cnts) {
    int flag = -1;
    int cnt =0;
    int ret;
    data recv;
    memset(&recv, 0, sizeof(recv));

    while (1)
    {
        /* Receive message from any process */
        if(flag != 0)
        {
            ret = MPI_Irecv(&recv, 1, mpi_data_type,
MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &requests[my_rank]);

            flag = 0;
        }
        MPI_Test(&requests[my_rank], &flag, &status[my_rank]);

        if (flag != 0)
        {
            if (ret == MPI_SUCCESS )
            {
                enum manageTag tag = status[my_rank].MPI_TAG;
                int source = status[my_rank].MPI_SOURCE;

                if (tag == mSNAPSHOT)
                {
                    printf("################ SAVING SNAPSHOT ###############");
                    printf("recv.custom_round_number: %d,
recv.round_number: %d, recv.value: %d\n", recv.custom_round_number,
recv.round_number, recv.value);
                    printf("################# END SNAPSHOT ################");

                }
                else if (tag == mREVERT)
                {
                    printf("################ DELETE SNAPSHOT ###############");
                    printf("PERFORM CUSTOM LOGIC FOR MANAGING SNAPSHOT");
                    printf("################# END SNAPSHOT ################");
                }

                cnt = increment_counter(telemetry_msg_cnts, 1);
            }
            flag = -1;
        }

        if (cnt>0)
        {
            if (!flag)
                MPI_Cancel( &requests[my_rank] );
                break;
        }
    }
}

// trigger a snapshot to saved data known as recv
trigger_snapshot(row_comm[TELEMETRY], recv, num_procs,
mpi_data_type, requests);

// retrieve snapshot
handle_snapshot_messages(row_comm[TELEMETRY], my_rank,
requests, status, mpi_data_type, telemetry_msg_cnts);
```

To run the program

```
$ mpicc single-paxos3-snapshot.c && mpiexec -n 5 ./a.out
```

4.5 Byzantine Protocol (BFT)

The resiliency discussed in SubSections 4.1, 4.3 would fail if there is $< \frac{1}{2}$ of the number of live processes. More robust algorithms can handle cases of failure exceeding the tolerance bound of traditional Paxos and raft algorithms. Some algorithms have mechanisms to handle cases where the failed process can take arbitrary actions such as sending bad messages and activating unnecessary state transitions, and even denial of service can affect a distributed system. It does not prevent some interruption, but it can mitigate the attack and ensure that an agreement can still be reached.

These algorithms with robust error handling included in the consensus algorithm are known as Byzantine protocols. These allow for building distributed systems that function in chaotic channels with significant disruption, as we can achieve a quorum with at least $\frac{1}{3}$ of the number of live processes. We consider scenarios with parallel leaders and also scenarios with a single leader serving as the proposer. There are several papers describing BFT [18, 41] with varying tolerance guarantees.

4.6 Distributed Commit and Transaction

Commit is the immutable operation that finalizes a transaction, making its modifications permanent and visible throughout the system. In contrast, a transaction represents a logical and atomic unit of work, potentially comprising multiple individual operations (reads, writes, updates, deletions). A transaction is the combined procedure for performing a task, consisting of a sequence of steps. Commit serves as the concluding step of a transaction, confirming its successful execution and guaranteeing the durability of all associated changes. Transactions prevent race conditions, inconsistent updates, and conflicting operations, and can provide strict guarantees on the execution plan.

Atomic commit requires that a collection of processes jointly decide whether a transaction is committed or aborted. The properties of a transaction include:

- Atomicity: a transaction is either fully committed or rolled back.

- Consistency: the last write is the latest read.

- Isolation: no interaction between processes.

- Durability: committed transactions are saved for later access.

We discuss two categorizations of transactions, as shown below:

- Flat transaction is a set of sequential actions synchronized across nodes.
- Nested transaction has a hierarchical structure where the transaction can trigger a set of sub-transactions.

Let's set up a simple, single-system banking scenario with two accounts (A and B) to prepare for our discussion of distributed protocols in SubSection 4.6.1. We will illustrate a basic transfer using a flat transaction (Figure 4.7) and a more complex breakdown using a nested transaction (Figure 4.8).

Figure 4.7 depicts a basic banking transaction, such as a fund transfer. It outlines the sequential operations involved and the two potential outcomes: successful completion (Commit) or failure necessitating a rollback (Abort) to maintain account integrity. For instance, insufficient funds in Debt Account A during a debit would likely lead to an Abort, a scenario covered by the "Failure" path originating from Credit Account B.

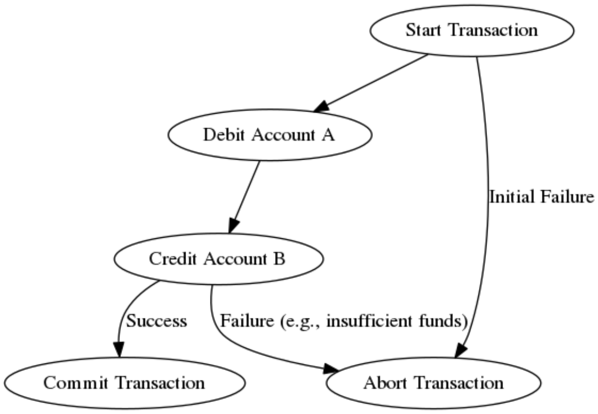


Figure 4.7: Transaction showing deposit and withdrawal.

Figure 4.8 presents a visual representation of a banking fund transfer implemented as a nested transaction, where the main operation is divided into a series of interdependent sub-operations. The overall process begins with a "Start Transaction" action, which in turn triggers the first sub-transaction, labeled "Start Sub-Transaction: Verify Funds." The purpose of this initial sub-process is to confirm the availability of adequate funds through a ”CheckBalance” operation. Based on the outcome of this check, the system either proceeds to a ”SufficientFunds” state, culminating in a successful ”Commit Sub-Transaction: Verify Funds,” or encounters ”InsufficientFunds,” leading to an ”Abort Sub-Transaction: Verify Funds.”. The subsequent step, ”Start Sub-Transaction: Transfer Funds,” is initiated solely upon the successful completion (commit) of the ”Verify Funds” sub-transaction. This second sub-process involves the core actions of ”Debit Account A” and ”Credit Account B.” A successful execution of both these actions results in a ”Commit Sub-Transaction: Transfer Funds.” Conversely, any failure during either the debit or credit operation leads to an ”Abort Sub-Transaction: Transfer Funds.”. The success of the entire transaction, spanning from the initial ”Start Transaction” to the final ”Commit Transaction,” is contingent upon the successful commitment of both constituent sub-transactions. If either the ”Abort Sub-Transaction: Verify Funds” or the ”Abort Sub-Transaction: Transfer Funds” occurs, the entire operation is rolled back, resulting in an ”Abort Transaction.” This mechanism ensures the integrity of the data by reverting any partial modifications. Consequently, the final status of the primary transaction is directly determined by the combined outcomes of its sub-transactions.

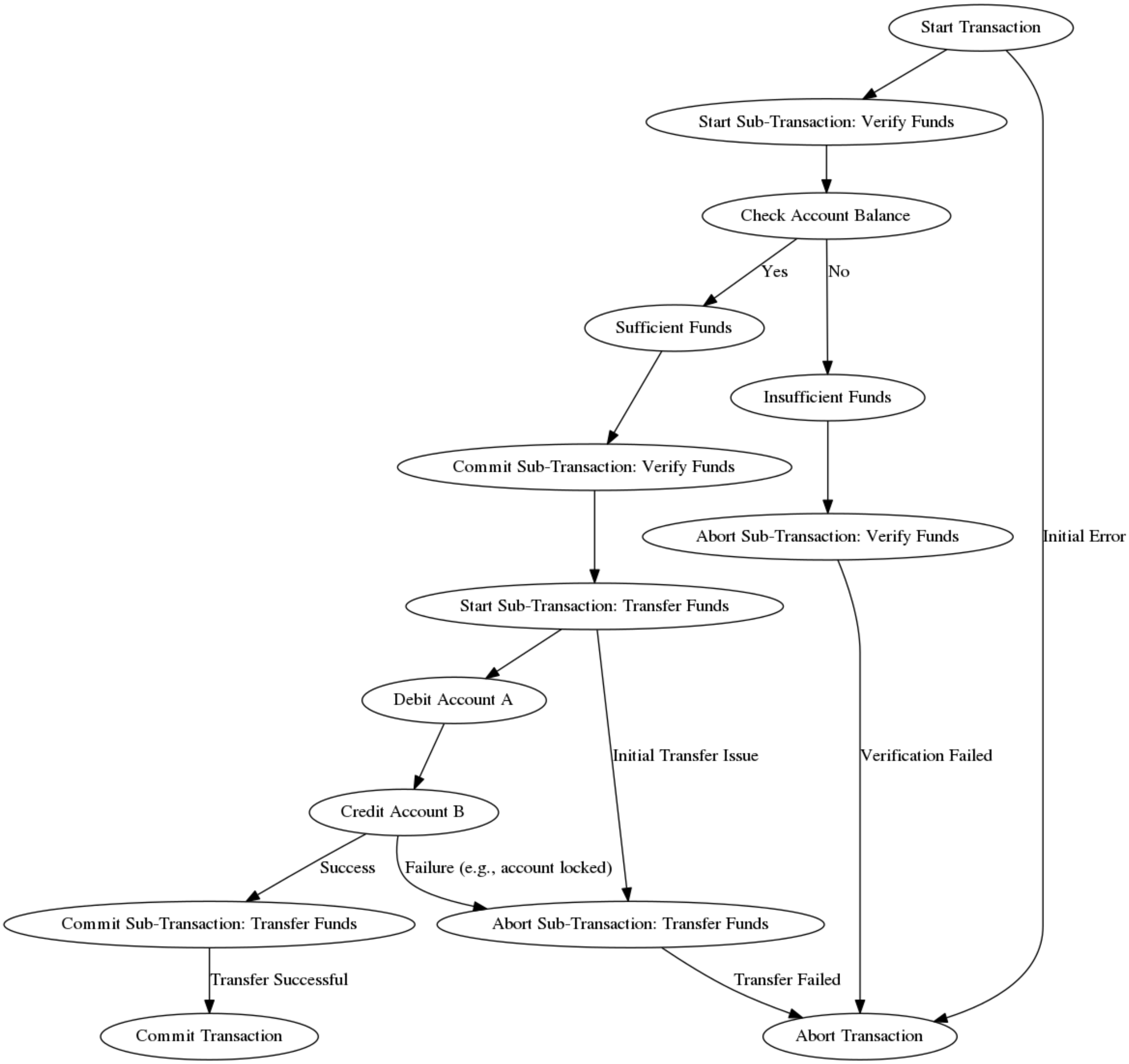

Start Transaction
Start Sub-Transaction: Verify Funds
Check Account Balance
Yes
No
Sufficient Funds
Insufficient Funds
Commit Sub-Transaction: Verify Funds
Abort Sub-Transaction: Verify Funds
Initial Error
Start Sub-Transaction: Transfer Funds
Debit Account A
Initial Transfer Issue
Verification Failed
Credit Account B
Success
Failure (e.g., account locked)
Commit Sub-Transaction: Transfer Funds
Abort Sub-Transaction: Transfer Funds
Transfer Successful
Transfer Failed
Commit Transaction
Abort Transaction

Figure 4.8: Nested Transaction containing sub-transactions depicting a bank transaction.

We aim to perform transactions between a client and a server. The client begins a transaction that must be executed on the server. In the case of a one-phase commit, which is unidirectional, the server cannot solely decide to abort a transaction during a client request. Due to the lack of concurrency control, if the server aborts the client's request, then the client will be unaware of the server's action. The client has to make another call to know the state of the server. An improvement is to introduce a coordinator who is always alive to help serialize the actions between the client and the server. This improvement has led to the development of several atomic control protocols in SubSection 4.6.1.

4.6.1 Atomic Commit Protocols

Two-Phase Commit

Two-Phase Commit (2PC) protocol permits any party (participant) can decide to roll back its transaction (due to failure) and the entire transaction is globally reverted [20, 43]. This requires a coordinator to facilitate concurrency control. The algorithm has two phases [20]: in the first phase, each party votes to commit or abort a transaction. Once, a party has voted to commit a transaction, it is binding and immutable. After voting each party goes to the ”prepared” state. In the second phase, each party in the transaction jointly executes the decision. If any party aborts or fails, then the overall transaction is aborted. The two phases ensure that all parties reach the same decision on committing or rolling back the transaction.

We describe the Two-Phase Commit Protocol depicted in Figure 4.9 as follows:

- The coordinator makes a ’canCommit’ request to every participant, and the participants respond with a vote. Only participants that send a ’yes’ transition to the ’prepared’ state.

- Upon receipt of the votes from the participants, if every vote is a ’yes’, then the ’doCommit’ request is sent to every participant. However, if any vote is a ’no’, then the current transaction on the coordinator is aborted, and then the ’doAbort’ command is sent to every participant.

- Participants, on receiving the ’doCommit’, proceed to commit the transaction; otherwise, if ’doAbort’ is received, then the transaction is aborted. The participant may send a ’haveCommitted’ request to the coordinator to ensure the information on the coordinator can be deleted.

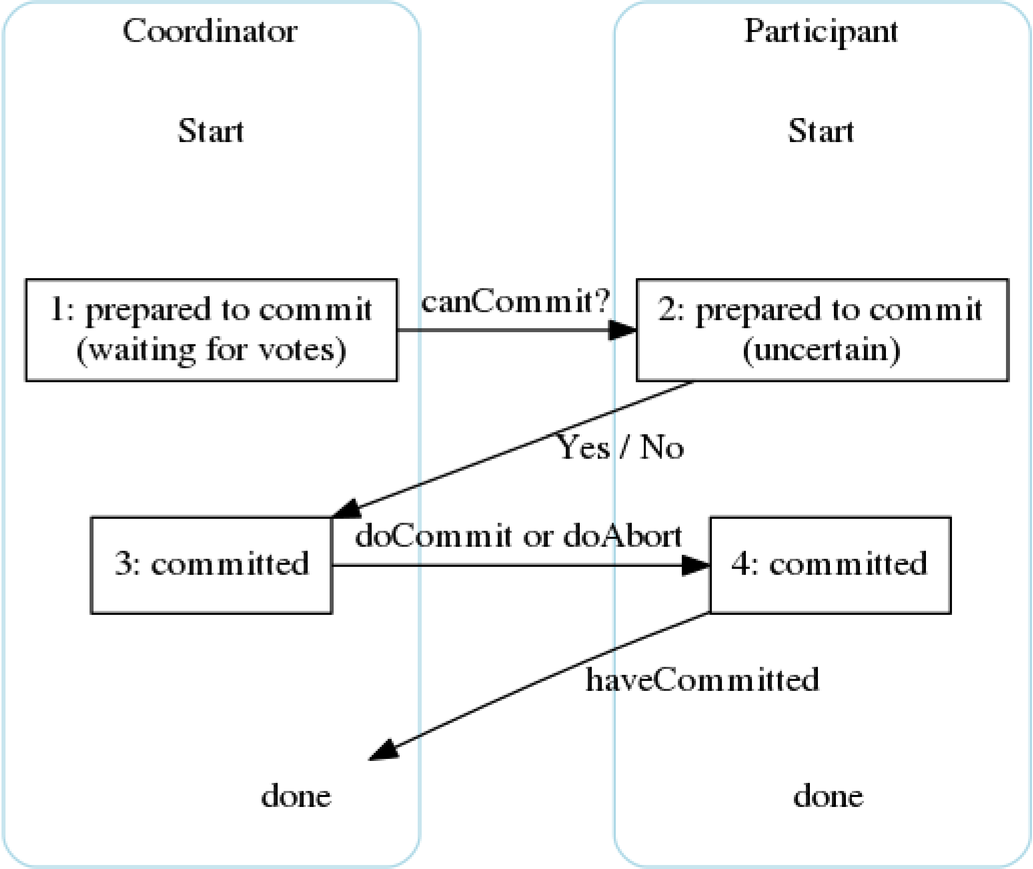


Figure 4.9: Two-Phase Commit [20].

Three-Phase Commit

Two-Phase Commit fails when the coordinator is dead and participants in the transaction cannot determine the state of the transaction resulting in longer polling intervals as parties try to connect and get information from the coordinator. Three-Phase Commit solves the problem by taking more rounds [20, 43] as we perform a precommit round and obtain ACKS before proceeding to commit.

We describe the Two-Phase Commit Protocol depicted in Figure 4.10 as follows:

- The coordinator makes a 'canPreCommit' request to every participant, and the participants respond with a vote. Only participants that send a 'yes' transition to the 'prepared' state.

- Upon receipt of the votes from the participants, if every vote is a 'yes', then the 'doPreCommit' request is sent to every participant. However, if any vote is a 'no', then the current transaction on the coordinator is aborted, and then the 'doAbort' command is sent to every participant.

- The participants, after precommitting, send an ACK to the coordinator. If there are no negative ACKs or missing ACKs, then the 'doCommit' command is sent to every participant.

- Participants, on receiving the 'doCommit', proceed to commit the transaction; otherwise, if 'doAbort' is received, then the transaction is aborted. The participant may send a 'haveCommitted' request to the coordinator to ensure the information on the coordinator can be deleted.

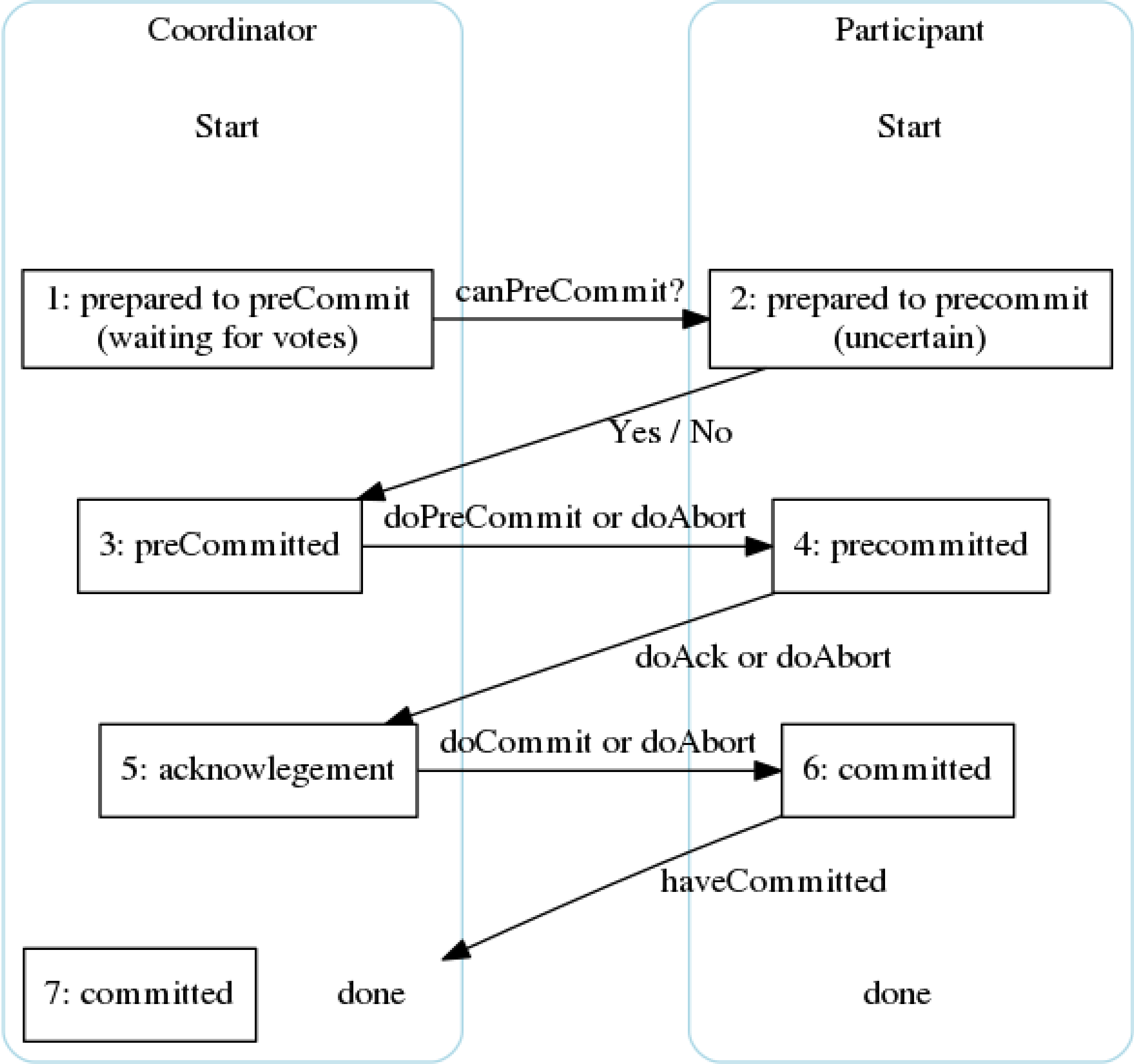


Figure 4.10: Three-Phase Commit [20].

4.7 Routing Algorithm

For practical purposes, a thorough understanding of the following concepts may be necessary.

- Deep understanding of L2 / L3 networking
- IP routing protocols such as RIP, OSPF, BGP, IS-IS, PIM.

- Layer 2 such as 802.1d 802.lax spanning tree protocol, 802.1AB link aggregation control protocol, link layer discovery protocol, RFC 1812 IP routing.

- Ethernet bridging and routing in Distributed software.

- Wireless protocol (802.11, Bluetooth, Zigbee).

- Creating a testbed (mini-internet) with the (backbone, router, switches) for testing routing protocols from scratch. Make use of dockers and VMs as shown in Network Programming in Python book.

- Do experiments with network shaping in Docker. See `https://github.com/lukaszlach/docker-tc`.

Chapter 5

Anti-Entropy Techniques

"One of the truest tests of integrity is its blunt refusal to be compromised. – Chinua Achebe"

"Nothing is more important than the state of the soul." – G.K. Chesterton

Entropy in distributed systems signifies a state of inconsistency or disagreement among the data held by different nodes. Even after a consensus algorithm has theoretically concluded and terminated, various real-world factors can lead to some nodes becoming out of synchronization with others. These contributing factors encompass the inherent complexities of distributed networks, including unpredictable network topologies, variable and often non-deterministic message delivery times, arbitrary ordering of message arrival, and the ever-present possibility of node failures. Effectively addressing these inconsistencies is paramount for maintaining the data integrity, reliability, and overall correctness of distributed systems.

The process of resolving these discrepancies can be conceptually viewed as a form of "secondary consensus" that occurs after the initial "primary consensus" protocol has aimed to establish agreement on a particular value or state. It also closely relates to the principle of eventual consistency, where updates to the system's state might not be instantaneously reflected across all nodes but are expected to propagate and converge to a consistent state over a certain period. However, in practical scenarios, this eventual convergence might take an unacceptably long duration, necessitating the implementation of active mechanisms to expedite the synchronization process and ensure timely agreement.

The robustness and overall effectiveness of a distributed software system can often be directly attributed to how well it proactively handles these inherent inconsistencies. While the specific implementation details and proprietary techniques might vary significantly between different systems, the general frameworks for reasoning about and systematically addressing entropy are based on well-established principles and algorithmic patterns.

One fundamental approach to mitigating these issues involves the deployment of background processes that periodically perform checks for discrepancies in the state held by different nodes. Upon detecting an out-of-sync condition, these processes can automatically trigger retry mechanisms to re-request missing or potentially corrupted data or employ sophisticated error correction techniques to repair incomplete or inconsistent data, thereby facilitating conflict resolution and restoring a consistent state. A straightforward yet often effective conflict resolution strategy is for a node that detects an inconsistency to request the correct or missing data from its neighboring nodes and subsequently updates its local state upon successful retrieval during these retry attempts. Well-known examples of such proactive synchronization techniques include read repair, where inconsistencies are actively detected and fixed during read operations initiated by clients, and hinted handoff, a mechanism where a temporarily available neighbor node accepts and temporarily stores writes destined for a failed node, ensuring that these writes are delivered once the intended recipient recovers and rejoins the system.

5.1 Gossip Algorithm

The Gossip protocol [25, 29, 30, 32] presents a robust and widely applicable anti-entropy mechanism that can be effectively employed in distributed systems to promote data consistency [37]. This protocol leverages a probabilistic approach to disseminate information (messages) across a network, drawing a compelling analogy to the way rumors or information propagate through a social network. A significant advantage of the Gossip algorithm is its inherent fault tolerance and resilience to failures, even in the face of substantial and unpredictable network disruptions or node outages. While the epidemic approach, characterized by the rapid and potentially ubiquitous spread of information to all reachable nodes, is a common and intuitive implementation strategy, Gossip protocols can also be implemented using non-epidemic strategies that strategically leverage overlay network structures or maintain partial views of the system's membership to optimize message dissemination and resource utilization.

5.2 CRDT

Conflict-free Replicated Data Types (CRDTs) represent another powerful and increasingly popular class of anti-entropy mechanisms in distributed systems. Riak, a distributed NoSQL database, was a notable pioneer in the widespread adoption and practical application of CRDTs [5, 37]. A CRDT is a specialized type of data structure meticulously designed to maintain a consistent state across multiple independent replicas, regardless of the order in which concurrent operations are executed on those individual replicas. This inherent order-independence property makes CRDTs exceptionally well-suited for scenarios involving collaborative applications and seamless remote synchronization (replication) of data across numerous geographically distributed devices, a task that can be inherently complex to manage for data consistency using traditional concurrency control mechanisms. Mathematically, CRDTs can be formally characterized as partially ordered monoids that possess a well-defined lattice structure. To ensure their conflict-free nature, they must satisfy the algebraic properties of commutativity (the order in which operations are applied does not affect the final state) and idempotence (applying the same operation multiple times has the same effect as applying it only once).

CRDTs offer several compelling advantages for building highly available and eventually consistent distributed applications:

- **Eventual consistency:** They provide a strong guarantee that all replicas of the CRDT will eventually converge to the same consistent state once all locally generated operations have been successfully propagated and applied to all other replicas in the system.

- **Preserve ordering of the data:** While the order of concurrent *operations* might not be critical for achieving eventual consistency, certain types of CRDTs are specifically designed to inherently preserve the logical or intended ordering of the *data* they are managing, which is crucial for applications requiring ordered data structures.

- **Local-first application:** Operations performed on a CRDT can be applied locally on a replica immediately without the need to wait for synchronization with other replicas. This ”local-first” characteristic significantly improves the responsiveness and perceived performance of applications, leading to a better user experience, especially in high-latency network environments.

A diverse range of CRDTs has been developed, each tailored to manage different types of data and support specific sets of operations. Common examples include map CRDTs (such as the Last-Write-Wins Map - LWW-Map), set CRDTs (such as the Grow-Only Set - G-Set), and many other specialized data structures designed for various consistency and concurrency control requirements [5, 37].

We have implemented a version of LWW-Map is shown in lamport1-majority-voting8.c. Basically, we make updates and use the largest timestamp of processes to determine the final value of a map.

The provided C source code implements a Grow-only Counter (GCounter) as shown in crdt-gcounter.c, a type of Conflict-free Replicated Data Type (CRDT), utilizing the Message Passing Interface (MPI) for parallel operation. Each parallel process maintains a local state stored in the ‘GCounter‘ structure, where the ‘counts‘ array tracks the contributions of every process on a per-index level. The core CRDT operation is the merge, which is performed using the collective primitive ‘MPI_Allreduce‘ with the reduction operator ‘MPI_MAX‘. This operation ensures that every process receives the global state where each element of the merged array is the maximum of the corresponding local element from all contributing arrays, correctly satisfying the GCounter’s merge property: $merge(A, B) = max(A_i, B_i)$. Finally, the ‘gcounter_value‘ function calculates the counter’s total by performing a summation over all indices of the merged array; while the true CRDT merge logic uses ‘MPI_MAX‘, the summation provides a shorthand method to check that the merged arrays have similar content and allows for a final, consistent global value check across all processes for demonstration purposes.

lww-map.c implements a Last-Write-Wins Map (LWW-Map), a type of Conflict-free Replicated Data Type (CRDT), using the Message Passing Interface (MPI) to handle state replication and global merging across multiple parallel processes. The basis of the code is to allow independent, concurrent updates (puts and deletes) to a key-value store on different nodes, with a deterministic merge function that resolves conflicts using a logical timestamp and a node ID tie-breaker. The core data structure, ‘LWWMap‘, holds an array of ‘Entry‘ structs, where each entry records the key, value, a logical timestamp (‘ts‘), the node ID (‘node‘) that made the update, and a tombstone flag (‘tomb‘) to mark deletions. The ‘put‘ and ‘del‘ functions increment the map’s local clock ($m \to clock$) before writing the entry, ensuring newer operations have a higher timestamp. The critical functionality is handled by the ‘global_merge‘ function: Rank 0 first collects the local state arrays from all other ranks using ‘MPI_Gather‘ to determine the size and subsequent ‘MPI_Recv‘ (or local ‘memcpy‘ for itself) to gather all individual entries into a single large buffer. Rank 0 then performs the actual merge logic: it iterates through the collected entries and for each unique key, it compares the timestamp and node ID of the new entry against the currently merged entry using the ‘newer‘ function, keeping only the one with the later timestamp or higher node ID. Finally, Rank 0 broadcasts the resulting, fully merged ‘LWWMap‘ state back to all other processes using ‘MPI_Bcast‘, ensuring every node is synchronized. The ‘main‘ function demonstrates a conflict scenario where different ranks update or delete the same keys (’a’, ’b’, ’c’), followed by a global merge and a call to ‘print_visible‘, which displays the final, consistent state across all ranks, omitting any entries marked by a tombstone.

5.3 Operational Transformation

Operational Transformation (OT) is a sophisticated technique primarily employed in collaborative editing systems, such as shared document editors, to maintain consistency across multiple concurrent users editing the same document simultaneously. Similar to CRDTs, OT aims to resolve conflicts that arise from concurrent operations performed by different users by transforming the semantics of these operations based on the context of previously executed operations. However, OT typically deals with ordered and sequential operations on linear data structures like text or rich text documents, and its underlying mathematical principles and implementation complexities differ significantly from those of CRDTs, which are generally more focused on unordered sets or map-like data structures.

TODO: PROVIDE IMPLEMENTATIONS

5.4 Ancillary Structures

Ancillary data structures, while not directly mechanisms for achieving consensus or propagating updates, can play a vital role in efficiently verifying whether the state held by different nodes in a distributed system is in disagreement or has become inconsistent. Structures like the Merkle tree provide an efficient method for identifying and potentially facilitating the resolution of such conflicts. A Merkle tree is a tree-like data structure where each non-leaf node is a cryptographic hash of its child nodes, and the leaf nodes are cryptographic hashes of individual data chunks or blocks. This hierarchical hashing mechanism allows for the efficient and secure verification of data integrity and consistency across distributed systems. We can effectively identify disagreements among nodes if they fail to produce matching results during audit or consistency proofs based on their respective Merkle trees. Let us delve into the details of these proofs:

- **Audit proof:** An audit-proof enables a node to efficiently and cryptographically verify that a specific, single data chunk exists within a Merkle tree held by another node, without the necessity of downloading the entire potentially large dataset. This is achieved by the requesting node providing the path of hashes from the leaf node corresponding to the data chunk up to the root of the Merkle tree, allowing the verifying node to recompute the root hash and compare it with its own.

- **Consistency proof:** A consistency proof allows a node that holds one version of a Merkle tree (representing the state of the data at a particular point in time) to efficiently and cryptographically verify that another node holding a potentially different version of the same Merkle tree represents a consistent historical version of its own tree. This enables efficient comparison of different states of the same dataset over time, identifying points of divergence or ensuring that one version is indeed an ancestor of another.

Consistency verification using Merkle trees significantly facilitates:

- **Data verification:** Ensuring the integrity and cryptographic correctness of the data held by individual nodes by verifying the consistency of their Merkle trees or specific branches thereof.

- **Data synchronization:** Efficiently identifying the specific data chunks or subtrees that differ between nodes by comparing their Merkle tree structures, allowing for targeted synchronization of only the differing data and minimizing the overhead of transferring large, identical datasets.

My understanding of how the Merkle tree functions is that it provides a highly efficient and cryptographically sound way to pinpoint the precise data chunks or segments that exhibit differences between nodes in a distributed system. A common initial step is for nodes to exchange the root hash of their respective Merkle trees as lightweight metadata. If the root hashes match, there is a very high probability (depending on the cryptographic hash function used) that the entire underlying dataset is consistent across the nodes. However, if the root hashes differ, it definitively indicates a mismatch in the data. Given the hierarchical structure of the Merkle tree, it allows for a more granular and efficient comparison process. Nodes can recursively traverse down the tree, comparing the hashes of child nodes at each level. When a hash mismatch is detected at a particular level, it indicates that the inconsistency lies within the subtree rooted at that node. This allows nodes to selectively request and synchronize only the specific sub-items or data chunks that are inconsistent, significantly reducing the network bandwidth and processing overhead compared to transferring and comparing entire datasets.

The intriguing and often debated aspect is the claim that Merkle trees can, in certain specific scenarios, facilitate conflict resolution directly, going beyond simply identifying discrepancies and triggering subsequent retry or synchronization mechanisms. While my intuition aligns with the understanding that conflict resolution typically necessitates a separate, higher-level mechanism to decide which version of conflicting data to ultimately retain or merge, there might indeed exist specific scientific papers or specialized techniques that leverage the inherent structural properties or metadata embedded within Merkle trees to automate certain limited types of conflict resolution, particularly in specific data models or application contexts. I would be very interested in learning about such scientific papers that rigorously detail methods for automatic conflict resolution directly derived from the structure or properties of Merkle trees, rather than just using them for efficient detection of inconsistencies.

The fundamental challenge of resolving conflicts in distributed systems often involves navigating the inherent trade-off between achieving strong consistency and maintaining high availability.

- **Strong consistency:** This stringent approach ensures that every node in the distributed system is effectively locked or prevented from serving potentially stale data until all nodes have successfully acknowledged and applied an update to a new value. This guarantees that all read operations will reflect the most recent write operation, providing a linearizable view of the data. However, achieving strong consistency can significantly impact the system's availability and performance, especially in the presence of network partitions or transient node failures, as the system might become unavailable if a quorum of nodes cannot be reached.

- **Eventual consistency:** This more relaxed consistency model represents a deliberate trade-off that prioritizes high availability and partition tolerance over immediate consistency. In eventually consistent systems, updates are applied to some replicas, and the system provides a probabilistic guarantee that all replicas will eventually converge to the same consistent state at some point in the future, assuming no further updates occur. During the period before complete convergence, different replicas might temporarily serve different (potentially stale or outdated) data. A common and often desirable consistency guarantee within eventually consistent systems is ”read-your-writes” consistency, which ensures that a client will always see the results of its own immediately preceding write operations, even if other replicas in the system have not yet fully processed the update.

5.5 Error Correction Code

Failures are an inherent and unavoidable aspect of operating large-scale distributed systems, and the field of coding theory provides a rich and well-established set of mathematical schemes and algorithms for detecting and recovering from these errors, thereby ensuring data integrity, reliability, and continuous availability. We will focus on two prominent and widely used schemes for error recovery in the following subsections: Erasure Coding (Subsection 5.5.1) and Multiple Description Coding (Subsection 5.5.2).

For readers interested in a more in-depth exploration of the theoretical underpinnings and practical applications of various error recovery codes, the following highly regarded texts are recommended as valuable resources:

- Error Correction Coding: Mathematical Methods and Algorithms by Todd K. Moon
- Fundamentals of Error-Correcting Codes by W. Cary Huffman and Vera Pless
- Turbo Code Applications: A Journey from a Paper to Realization by Keattisak Sripimanwat

5.5.1 Erasure Coding

Erasure coding (EC) is a sophisticated and highly effective redundancy scheme specifically designed to enhance data reliability and provide robust fault tolerance, particularly in distributed storage systems where multiple storage drives or nodes can fail unexpectedly and concurrently. This technique offers a significantly more storage-efficient alternative to traditional full replication strategies, such as those used in conventional RAID (Redundant Array of Independent Disks) configurations, by allowing for a configurable and often substantially lower storage overhead while still providing comparable or even superior levels of data durability and availability.

The fundamental principle behind erasure coding involves systematically organizing data into a set of data segments or blocks and then generating a corresponding set of redundant parity blocks. These parity blocks contain carefully encoded complementary information derived from the original data blocks, which can be mathematically leveraged to reconstruct the original data in the event of the loss or corruption of some of the data blocks or parity blocks themselves. An Erasure Coding algorithm intelligently utilizes the remaining intact data blocks and the parity blocks to both detect the occurrence of failures (erasures) and, crucially, reconstruct the lost data, ensuring data integrity and availability. It is generally recognized as impractical and highly wasteful of storage capacity to maintain a dedicated parity block for each original data block. Instead, a common and more efficient approach is to group a set of n original data blocks together and generate a smaller set of m parity blocks that collectively provide the necessary redundancy for error recovery. For such an (n, m)

erasure coding configuration to function effectively in a real-world storage system, a total of $n + m$ storage drives or nodes are required. The key advantage is that the storage overhead is determined by the ratio of m to n, which can be significantly lower than the $1 : 1$ overhead of full replication while still offering substantial fault tolerance (the ability to recover from the loss of any m drives).

5.5.2 Multiple Description Coding

Multiple Description Coding (MDC) represents another important class of redundancy schemes specifically designed to enhance resilience to failures and mitigate the impact of network issues by strategically splitting a single source data stream into multiple, inherently redundant substreams or descriptions. These individual segments or descriptions might be encoded with varying levels of detail, and resolution, or possess distinct characteristics tailored for different transmission or reception conditions, but they all contain a similar core of essential information about the original data. The receiver (callee) then employs a variety of techniques, often dynamically adapting based on real-time quality of service (QoS) metrics, to selectively utilize the available received segments for reconstructing the original data stream.

A prominent and widely deployed real-world example of Multiple Description Coding principles in action is adaptive bit rate (ABR) streaming for video content delivery over the internet. In ABR, a single video is encoded into multiple independent streams with varying levels of quality and resolution (e.g., low, medium, high definition). The video player (callee) continuously monitors prevailing network conditions, such as the available internet speed, network bandwidth, and the level of contention on the network path. Based on these dynamically measured QoS metrics, the player intelligently and seamlessly switches between requesting and decoding segments from the different resolution streams. This adaptive approach allows the player to provide the best possible viewing experience under the current network conditions, gracefully degrading the video quality if the network deteriorates and automatically improving it when the network capacity allows.

[1]

[1] See an illustrative example of optimizing a distributed data store for operation at significant scale in the Uber blog

Chapter 6

Peer-to-Peer Computing

"In the midst of chaos, there is also opportunity." – Sun Tzu

"Life is a wheel of fortune and it's my turn to spin it." – Tupac Amaru Shakur

In the peer-to-peer (P2P) paradigm, each network node functions as both a client and a server simultaneously. This allows any node to communicate directly with its neighbors without the necessity of a dedicated central server. This decentralized arrangement inherently avoids a single point of failure, enhancing the system's robustness. However, some hybrid peer-to-peer systems exist where a central server is utilized for initial orchestration and connection establishment. These servers store information about the peers present in the network, facilitating the discovery of other nodes.

In contrast, the traditional client-server paradigm restricts direct communication between clients. For one client to interact with another, it typically needs to update the state on the central server, which then allows other clients to connect to the server and retrieve the shared information. In this model, the server becomes a critical component for all communication, making it a significant single point of failure for the entire system.

Peer-to-peer computing offers notable benefits, including reduced storage and computational costs, achieved by distributing the burden of computation and data storage across several participating nodes. The applications of peer-to-peer computing span a wide range of domains, including distributed file systems (e.g., OceanStore [31], PAST [24], CFS [21]), file sharing networks (e.g., BitTorrent [1], FastTrack [2]), and large-scale Grid computing platforms [3].

We present a peer-to-peer network of 4 nodes (peers) named "A", "B", "C", and "D" as shown in Figure 6.1. Each node can connect with any other node in the network. There is no central server or intermediary required for one peer to interact with another node resulting in decentralized network.

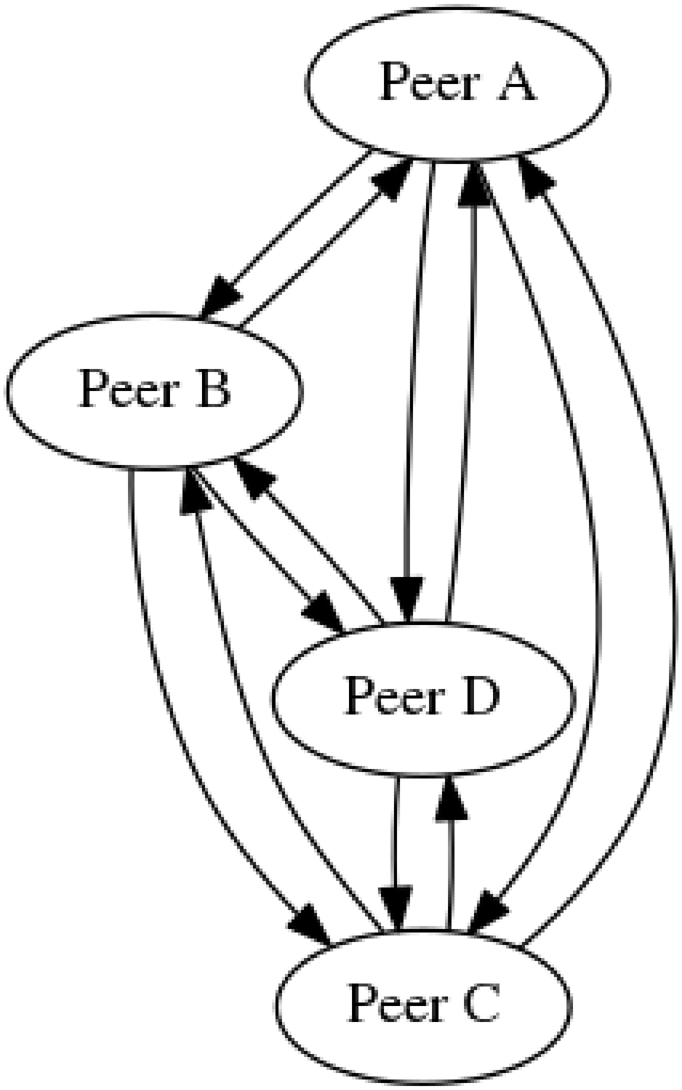


Figure 6.1: Network of nodes in a peer-to-peer network.

@TODO: IMPLEMENTATION

We will now discuss the categorization of peer-to-peer systems to address core concepts such as overlay networks, unstructured P2P systems, and structured P2P systems, as covered in Sections 6.1, 6.2, and 6.3 respectively.

6.1 Resilient Overlay Networks

An overlay network provides a dynamic, just-in-time network infrastructure built on top of an existing network layer (like the Internet) to facilitate decentralized communication in a scalable manner. It achieves this by implementing its own routing and packet transfer functionalities at the application level, abstracting away the complexities of the underlying physical network [16, 22].

Overlay networks must be designed to handle a high rate of client disconnections, which can occur due to various factors such as poor network conditions, limited available bandwidth, and the need for dynamic balancing of processing and bandwidth capacity across the participating nodes [17]. Consequently, there is a significant need for the development of resilient overlay networks that enhance the reliability of the network and make it robust to node and link failures [22]. Nodes within resilient overlay networks often incorporate self-healing capabilities. Such nodes can perform quality of service (QoS) checks on the communication links they use and dynamically re-route packets as necessary to circumvent failures and reduce latency.

A key characteristic of overlay networks is that the participating nodes do not need to reside within the same local network. They can be geographically dispersed across various administrative domains and networks. Due to the dynamic redirection of packets on the fly, the maintenance of a routing table at the overlay level becomes essential.

When communicating across different Internet Service Providers (ISPs), one might naively assume that Border Gateway Protocol (BGP) routing records are consistently up-to-date and reflect the optimal paths. However, this is often not the case, as the entries in an ISP's routing table can be influenced by a multitude of factors, including financial agreements between ISPs, the time it takes for BGP to converge after a routing update, and even political considerations [22].

The routing table maintained at the application level by the overlay network can potentially identify and utilize more efficient and reliable communication links compared to the underlying BGP routes. Content Delivery Networks (CDNs) are a prominent example of how overlay networks are leveraged to improve offsite caching and the efficient delivery of web resources to end-users [22].

6.2 Unstructured P2P Systems

Unstructured overlay networks are characterized by the absence of a predefined, deterministic structure for organizing nodes and data. Consequently, they do not inherently provide efficient query-matching mechanisms for locating specific data within the network [17]. Examples of P2P systems that utilize unstructured overlays with a random graph topology include Gnutella [44], the now-defunct Kazaa, and Freenet [19]. In such networks, a typical search query is flooded across the network to a subset or all neighboring nodes. Each node receiving the query then compares it against the metadata or content it holds.

6.3 Structured P2P Systems

In contrast to unstructured P2P systems, structured overlay networks impose a specific, well-defined structure on the network topology to facilitate efficient query matching and data discovery [17]. This efficient search capability is achieved by enforcing constraints on how the network is constructed and how data is indexed and located. In a structured P2P system, each node is typically associated with a unique key or identifier, and the actual data or metadata is stored at the node whose key is closest to the key of the data item. Searching for a specific data item involves querying the network using its associated key. The network's underlying structure, often based on sophisticated data structures like Distributed Hash Tables (DHTs), ensures that queries can be routed efficiently to the node(s) likely to hold the desired information. Prominent examples of P2P systems that leverage structured overlay networks include Chord [42], Tapestry [49], Kademlia [34], and Pastry [39].

Chapter 7

Practical Formal Verification of Distributed Systems

"It is better to be roughly right than precisely wrong." – John Maynard Keynes

Distributed systems present significant testing challenges due to the multitude of realistic and abnormal scenarios they must handle. This complexity is amplified by the inherent mechanisms within distributed systems designed to detect various types of failures, which can range from transient issues to more persistent Byzantine faults. Effectively addressing these failures is a primary objective in designing robust distributed systems that prioritize fault tolerance as a core principle. However, the implementation of mechanisms to mitigate these failures often introduces increased complexity into the system's architecture. This added complexity can inadvertently conceal subtle bugs, which can have severe consequences in mission-critical applications. Formal verification offers a powerful approach to verify, at the design level, that these potential problems do not exist. Careful attention must be paid to ensure that there is no discrepancy between the system's implementation and its formal specification. Once a system has undergone and successfully passed formal verification, a higher degree of confidence can be placed in the absence of critical bugs within the system.

While formal verification offers strong guarantees, it can be an expensive undertaking, primarily due to the significant effort required to develop a precise formal specification and to rigorously prove that the software implementation adheres to this specification. However, a substantial number of bugs can be effectively caught through less formal but still valuable testing methodologies such as unit tests, integration tests, and fault-injection tests. These testing approaches can help to uncover subtle bugs and significantly improve the overall quality of the distributed software. Software testing, in general, is less formal compared to the rigorous mathematical proofs employed in formal verification. Unit tests should be strategically designed to identify potential failure modes within individual components and to achieve high coverage of the problem space related to these failures. Integration tests play a crucial role in verifying the behavior of the system as different nodes communicate with each other over a simulated or real network channel within a cluster. A particularly insightful type of integration test is the fault-injection test, where engineers intentionally introduce specific failure scenarios into the running system and carefully observe how the collaborating components behave under these adverse conditions. For instance, in a distributed data store, a fault-injection test might involve simulating a node failure to observe if there is any data loss or corruption. Metamorphic testing is another valuable technique that can enhance test coverage by leveraging known relationships between different inputs and outputs and checking for violations of invariants that should hold even in the presence of failures. Several notable examples of robust testing practices in the industry include Netflix Simian Army, a suite of tools designed to randomly introduce failures into Netflix's cloud infrastructure to ensure resilience; Jepsen, a rigorous testing framework for distributed databases that focuses on uncovering consistency violations under various fault conditions; and Verdi, a research project focused on the formal verification of distributed systems implementations.

Formal verification serves as an overarching term for a collection of techniques aimed at mathematically proving whether a system's design or implementation adheres to a precisely defined set of specifications. A specification is a written document that describes the intended behavior and properties of a system in an unambiguous manner. While human language is often verbose and can include more information than is strictly necessary to convey the core requirements, mathematics offers a more concise and precise language for describing most phenomena in the physical and computational worlds. Unfortunately, even mathematical proofs can sometimes rely on informal arguments expressed in human language. Mathematical logic, particularly propositional logic, and temporal logic, can help to eliminate this redundancy and imprecision, leading to more rigorous and unambiguous specifications and proofs, which form the basis of formal methods.

The primary goal of formal verification is to identify potential errors and flaws in the system's design early in the development lifecycle, based on the formal specifications. Therefore, the specification should be carefully derived from the core requirements of the system to be truly useful in detecting critical issues. The specification must be kept as simple and focused as possible, so that the constraints on the system's behavior can be clearly and precisely defined, thereby mitigating the risk of the "second-system effect" (where an over-engineered second attempt at a system introduces more problems than it solves). A useful rule of thumb is to exclude any information from the specification that does not directly contribute to understanding the fundamental behavior and properties of the system, as this extraneous information is less likely to reveal critical violations. Creating a precise and comprehensive specification is a significant challenge in software development. Careful consideration must be given to composing a realistic specification that accurately reflects the system's requirements and subsequently writing source code that provably satisfies this specification. Ideally, the effort involved in writing the formal specification should be relatively less than the effort required to write the actual source code that implements and verifies the specified behavior, to avoid doubling the programmer's workload. Temporal logic is a specific type of formal logic that is particularly well-suited for proving the liveness properties of concurrent and distributed systems, ensuring that the system will eventually reach desirable states.

7.1 Model-Level Verification

One prominent model-based formal verification scheme specifically designed for distributed systems was developed by the renowned computer scientist Leslie Lamport. This scheme is called the Temporal Logic of Actions (TLA+). TLA+ allows for the description of the system under scrutiny using a set of mathematical equations and logical predicates that capture its state transitions and behavior over time. Notably, TLA+ also possesses non-temporal (state-based) semantics, making it a versatile formalism suitable for specifying a wide range of distributed systems, from communication protocols to concurrency control algorithms. Several tools, such as the TLA+ Toolbox and the interactive proof assistant Coq, support the specification and verification of systems using TLA+.

7.2 Code-Level Verification

Code-level verification takes a different approach compared to traditional model-level formal verification, where an abstract model of the problem space is created and checked for violations of safety and liveness properties. Instead, code-level verification directly examines the source code of the system, often involving running MPI/C source code within a specialized verification scheduler. A significant challenge in verifying concurrent programs is the potentially exponential number of possible interleavings of the execution of parallel threads or processes. To address this, code-level verification techniques often employ partial order reduction (POR) algorithms to identify and eliminate redundant or equivalent interleavings, thereby significantly reducing the state space that needs to be explored [46].

A notable tool in the domain of code-level verification for distributed systems is the ISP Formal Verification Tool (though the provided text does not give a full name or link, "ISP" likely refers to a specific research project or tool). These types of tools often focus on detecting critical concurrency bugs such as deadlocks and runtime error violations directly within the source code.

Code-level verification aims to bridge the "gulf of execution" that can exist between abstract model checking and the behavior of actual running source code by directly targeting the implementation. This alignment increases confidence that the verification results accurately reflect the real system's behavior. The primary focus is often on analyzing the interleavings of parallel threads to uncover subtle concurrency-related bugs.

While code-level verification offers the advantage of directly analyzing the implementation, model-based verification can provide productivity gains by allowing engineers to create abstract models of their concurrent designs and verify their properties before writing the full production code. However, code-based verifiers may face limitations in handling the full expressiveness of temporal logic specifications compared to dedicated model checkers.

One such tool is Valgrind. We show how to use 3 tools associated with Valgrind. The following description is done with the assumption that the user has previously installed Valgrind on the system.

memcheck

We can analyze the memory usage and health of a any program say test5.c in our example, which utilizes the OpenMPI library. This assumes that Valgrind is already installed on the system.

```
mpicc -g test5.c -o test5
mpirun -np 4 valgrind --tool=memcheck --leak-check=full ./test5
```

The output is given as follows.

```
 Leak Summary (Printed at the End of the Program):

==<PID>== HEAP SUMMARY:
==<PID>==     in use at exit: <bytes> bytes in <blocks> blocks
==<PID>==   total heap usage: <allocs> allocs, <frees> frees,
<bytes> bytes allocated
==<PID>==
==<PID>== LEAK SUMMARY:
==<PID>==    definitely lost: <bytes> bytes in <blocks> blocks
==<PID>==  indirectly lost: <bytes> bytes in <blocks> blocks
==<PID>==    possibly lost: <bytes> bytes in <blocks> blocks
==<PID>==      still reachable: <bytes> bytes in <blocks> blocks
==<PID>==         suppressed: <bytes> bytes in <blocks> blocks
```

HEAP SUMMARY: Provides an overview of heap memory usage.

1. in use at exit: The amount of heap memory that was still allocated when the program terminated. A non-zero value suggests memory leaks.

2. total heap usage: The total number of allocation and deallocation operations and the total bytes allocated.

LEAK SUMMARY: Categorizes unfreed memory blocks:

1. definitely lost: Memory that your program allocated and then lost all pointers to. This is a clear memory leak that you should fix.

2. indirectly lost: Memory that is pointed to only by definitely lost memory. Fixing the ”definitely lost” blocks will usually also resolve these.

3. possibly lost: Memory that might still be reachable through some complex pointer paths, but Valgrind isn’t entirely sure. Investigate these, as they often indicate leaks.

4. still reachable: Memory that was allocated but never freed, but the program still holds pointers to it at exit. While technically not a leak in the strictest sense, it’s often good practice to free this memory as well. You can suppress these reports with –show-reachable=no.

5. suppressed: Leaks that were ignored due to entries in suppression files.

It is better to reduce the size and number of blocks that are ’definitely lost or indirectly lost’ as part of improving your source code.

Helgrind and DRD

Helgrind and DRD are primarily designed to analyze programs using POSIX threads (pthreads) for concurrency errors and violations. It can provide a sanity check for the OpenMPI program, but it was not originally designed for the message-passing paradigm of our underlying runtime.

```
mpicc -g test5.c -o test5
mpirun -np 4 valgrind --tool=memcheck --leak-check=full ./test5
```

To get better output, get separate logs from each running process using the command provided below.

```
mpirun -np 4 valgrind --tool=helgrind --log-file=helgrind.%p.log ./test5
mpirun -np 4 valgrind --tool=drd --log-file=drd.%p.log ./test5
```

Interpreting the Output from Helgrind and DRD:

1. Valgrind (Helgrind or DRD) will output any detected threading errors or potential deadlocks to the standard error (stderr).
2. Carefully examine the output. Look for reports of lock contention, potential deadlocks (in Helgrind), or data races (in DRD).
3. Remember that these tools are interpreting the MPI processes as independent programs that might be using internal threading within the MPI library itself. The deadlocks they detect might be related to these internal threads or might be misinterpretations of MPI communication.

Chapter 8

Miscellaneous

”Rarely affirm, seldom deny, always distinguish.” – Thomas Aquinas

”Any subject can be made interesting, and therefore any subject can be made boring.” – Hilaire Belloc

8.1 Accelerated Computing

There are several computing architectures where specialized processors are designed to relieve the burden of computationally intensive tasks. Custom accelerators are crafted to offload computationally expensive tasks in an optimized manner onto dedicated hardware. Accelerators are experiencing increasing adoption. For example, the Graphics Processing Unit (GPU) is optimized for rapid matrix-based operations and can function as a co-processor alongside the system’s main processor. Furthermore, specialized cryptographic accelerators exist, such as the Intel Advanced Encryption Standard Instructions (AES-NI), which are utilized by cryptographic libraries like OpenSSL and LibreSSL. Offloading specific tasks to these accelerators can be viewed as a form of distributed system operating between the main processor and these specialized hardware components.

8.2 Storage Systems

The Unix file system provides a convenient hierarchical structure for the storage of data in files and directories. It was not built by default to be fault-tolerant. However, using a RAID structure, where many disks work together as a replicated data store, can help to achieve resilience in the face of failure. There is also a problem related to the increased storage cost, where the metadata per directory becomes significant at petabyte scales and beyond. There is room for optimization, as the metadata has to be loaded into memory from the disk to access the file. Due to the locality of data, this is a significant bottleneck for scalability.

Every volume, despite its sophisticated architecture, includes the following core features: a data file (the actual stored data), an index file (a searchable data structure), a journal file (which allows for persistence between restarts), a checkpoint (a snapshot that provides safe points for recovery after failures), and a client (an application that has access to the file system). These features have various names across different file systems, such as HDFS [40], Google File System [26], and Tectonic [36]. Some file systems [38] allow for sequential modification to a disk in a log-like manner. Another way to categorize storage systems is as single-tenant (Haystack [15], F4 [35], HDFS [40]) or multi-tenant (Tectonic [36]). A number of these storage systems utilize variants of Paxos and Raft to achieve distributed consensus across volumes. For more information on distributed consensus algorithms, please refer to Chapter 4, Sections 4.1 and 4.3.

There is room for optimizations to improve storage, leading to the development of some distributed storage systems. Haystack [15] improves upon the Unix file system by saving metadata in memory, thereby making lookups faster, while reading the actual file saved in the volume involves disk operations, thereby increasing throughput. The metadata is spread over a wide area of memory, allowing for compact representation in the Haystack store, rather than each directory storing the metadata. However, it is necessary to ensure that there is sufficient memory to load a significant amount of the metadata for lookup purposes. When an object is saved in a volume in a Haystack store, it is replicated to many secondary volumes for redundancy. The Haystack directory maps each volume to the metadata to facilitate reading the actual file using the metadata. The Haystack cache is used to increase the lookup speed of popular objects. This is ideal for immutable blobs, hence it follows the philosophy of write once, read always, and rarely update.

Haystack is deficient in utilizing the access patterns of requests for objects, and even the Haystack cache is not sufficient for granular control of access patterns. Hence, F4 [35] was developed to allow for incorporating access patterns into the design of a store. Rather than a RAID-like replication scheme, F4 utilizes distributed erasure coding and Reed-Solomon codes to ensure smarter, lower-cost replication to several secondary volumes. Objects saved in the store begin as ”hot” and then ”warm” over time. The idea is based on the pattern that newer objects are more likely to be accessed, while older objects are less likely to be accessed. This concept is known as temperature zones. Volumes are structured in a way that once a memory limit for a volume is reached, it is closed for writing and becomes read-only. These volumes are organized into temperature zones to take advantage of access patterns. There are also many systems (Pelican [14], RADOS [48]) that allow for intelligent use of access patterns, and a system (CRUSH [47]) that provides the option to specify weights for access patterns. Tectonic [36] is an evolution of the Haystack storage system, as it introduced multi-tenant features. Each tenant has its own namespace and delegates metadata lookup to a dedicated key store.

8.3 Coding Philosophy

- Made use of mpi, so we don’t have to consider the low-level socket networking stack and their quicks, between IPV4, and IPV6.

- Our implementation of Distributed systems creates a set of processes for the client. This may not be standard in the Paxos algorithm. We abstract it in our source code for ease of use.

- Our philosophy has been to think locally and act globally. We do computations on the node and communicate with other nodes by messaging.

- Production-grade Distributed systems should follow the best Software engineering practices. We are not optimizing our source code for production, but pedagogy.

- Rather than communicating by sharing memory, it is better to share memory by communicating.

- For the Paxos algorithm when using Unix timestamps as the round number or ballots for their monotonically increasing properties, then a necessary prerequisite is to synchronize the time settings on at least the set of proposers, or across the cluster, but that may be unnecessary. This paradigm was heavily used in our implementations.

- Organizing the processes into groups with custom communicators. This allows for targeted synchronization for grouped processes without impacting the total processes in the application. This logic allows for precise control of a group of processes.

- When trying to create an array of atomic counters. It is desirable to utilize an array of shared pointers, rather than an array of shared values.

- We can cancel pending requests and tune the criteria for quorum based on business needs. This would impact how resilience of the Distributed Systems.

- We use pooling on receiving the message and checking each tag, rather than waiting on specific tags to make code modular. With my quest to go low level, rather than use C++, I tried objected-oriented design in C to enhance modularity but abandoned the idea due to loss of type safety.

- Always pool on waiting reads in a busy-wait style.

- Make use of a simple structure. Even our log for sequence Paxos is not a log, but an abused linked list with some atomic primitives.

- Our sequence Paxos uses a single Paxos on each item that the proposer will send. Unfortunately, our logic is restricting to only the possibility of having one proposer.

- Retrieving messages and probing to check the tag of messages to identify a specific event. Busy waiting is used to retrieve messages on an irecv. Otherwise, only the last sent is received on polling. This can be a bug where you retrieve the same message multiple times.

- It is good to take steps to avoid both deadlocks and livelocks. Deadlock can happen in a mismatched message order between send and receive especially in blocking mode. It is possible in non-blocking mode to consider how request objects are owned between the receiving and sending nodes.

- Livelock is possible too in a non-blocking case when we pull in a busy wait manner. We exit from the end of the loop when we have received the expected number of messages. It can be sensible to keep track of the number of exchanged messages to force an exit from the endless loop.
- There are problems with passing pointers across nodes. This is because we don't have a universal shared memory. Always keep pointers local as a lack of distributed memory makes indirection on a pointer useless.
- Use timeouts to prevent resources from waiting indefinitely for computing.

8.3.1 Review of Selected Source Codes

- leader-election3.c This is a simplified version of our adaptation of the bully algorithm where the first candidate becomes the node to call an election and choose itself. The rationale is that if the client can start an election, then it is alive.
- two-phase-commit.c

  The execution of the transactional functions happens at slightly different stages for the coordinator and the participants. The coordinator acts as the trigger and executes the "commit" action locally based on the votes, while the participants execute it upon the coordinator's command.

  Potential Issues:

  - Asymmetry in Commit Execution: While this model achieves the outcome of either all participating nodes (coordinator and participants) effectively committing or aborting, the timing and triggering of the actual transactional function execution are different. This might be a point of confusion or a simplification for the demonstration.
  - No Confirmation from Participants: The coordinator sends COMMIT but doesn't wait for any acknowledgment from the participants that they have indeed committed. In a more robust system, you might have a third phase (though the name "two-phase commit" implies two phases) or some form of confirmation to ensure all participants have completed the action.

  - Error Handling After Decision: If the coordinator successfully decides to commit and sends COMMIT messages, but one or more participants fail to execute the commit, the coordinator isn't aware of this failure in the current implementation.

- sequence-paxos4.c

  The C code implements the Sequence Paxos consensus algorithm using MPI to coordinate actions between processes acting as Clients, Proposers, Acceptors, and Learners. It's designed to agree on a sequence of values in a distributed system, maintaining order even with potential failures. The Client proposes a sequence of values, and the Proposers, Acceptors, and Learners work through the Paxos protocol for each value to achieve agreement. Learners inform the Client of decided sequence positions, ensuring the Client proposes the next value in the sequence only after the previous one is confirmed. This process continues until all values are agreed upon, demonstrating distributed consensus on an ordered sequence.

- lamport1-majority-voting8.c

  The C code simulates a distributed key-value store where processes use Lamport clocks and majority voting to handle concurrent updates. MPI is used for communication, and each process maintains a local copy of the key-value store. When a process wants to update a value, it sends messages to other processes containing the key, the value, and its Lamport clock. Processes update their local Lamport clocks based on received messages to track the order of events. Each process gathers messages and uses the Lamport timestamps to determine the most recent (agreed-upon) value based on a majority, effectively resolving conflicts in a distributed setting.

Defensive Programming

One good practice is to wrap function calls to facilitate debugging and resilience of OpenMPI program.

```
#include <mpi.h>
#include <stdio.h>
```

```
#define SAFE_MPI_CALL(call) \
    do { \
        int ret = (call); \
        if (ret != MPI_SUCCESS) { \
            char error_string[MPI_MAX_ERROR_STRING]; \
            int length_of_error_string; \
            MPI_Error_string(ret, error_string, &length_of_error_string); \
            fprintf(stderr, "MPI Error (%s:%d): %s\n", __FILE__, __LINE__, error_s
            MPI_Abort(MPI_COMM_WORLD, ret); \
        } \
    } while (0)

int main(int argc, char** argv) {
    SAFE_MPI_CALL(MPI_Init(&argc, &argv));
    int rank;
    SAFE_MPI_CALL(MPI_Comm_rank(MPI_COMM_WORLD, &rank));
    printf("Hello from rank %d\n", rank);
    SAFE_MPI_CALL(MPI_Finalize());
    return 0;
}
```

This code introduces a safety mechanism for MPI function calls through a macro named SAFE_MPI_CALL. When you wrap an MPI function call with this macro, it automatically checks if the function executed successfully. If an error occurred, the macro will fetch the specific error message from the MPI library, report it to the error output along with the exact location in your code where the failure happened, and then immediately stop the entire MPI program. This ensures that errors in MPI operations are clearly identified and prevent the program from continuing in a potentially unstable or incorrect state. Using this macro promotes more robust and easier-to-debug MPI applications by standardizing error checking. Unfortunately, there is an overhead introduced by the macros, but consistent error handling outweighs the potential downsides.

We have saved the best for the last. You must have noticed recurring themes in the book where we make abstractions. For example, our leader election uses a failure detector. Decomposing systems into components allows one to focus on a part of a problem at the time. People call it adding one more layer of indirection, it helps to reason about systems but avoid leaky abstractions at all costs. "Most problems in Computer science can be solved with another level of abstraction".

8.4 Case Studies

We describe our implementation of a barebone serverless framework for autoscaling in Subsection 8.4.1 and an overview of distributed computing patterns in Subsection 8.4.2.

8.4.1 Implementation of AutoScaling Framework (Serverless)

I've also developed a basic autoscaling framework that adds or removes compute nodes based on predefined CPU load thresholds. A drawback of this initial implementation is its non-incremental nature: we currently tear down existing nodes before adding new ones, which can lead to temporary memory spikes. Implementing autoscaling in OpenMPI requires careful engineering because the total number of MPI processes for communication groups must be set at the initialization. While OpenMPI groups are useful for limiting computation to specific processes (beneficial to avoid performance issues with too many communicating processes), this fixed-size requirement makes dynamic scaling difficult, and is why we've adopted our current node replacement strategy. See source code in autoscaling.py. The parameters for our implementations are arbitrarily chosen.

We detected CPU usage on each node using command top run using execl to run the top executable available in the PATH usually and use popen to read the content from top command. Replicate the CPU usage and estimate the average. Use as a heuristic to trigger process management.

Our implementation has Constants and helper functions.

Constants

- MIN_THRESHOLD = 25: The lower CPU utilization threshold. If the average

CPU load falls below this, the script will potentially reduce the number of MPI processes (though the current logic only increases).

- MAX_THRESHOLD = 75: The upper CPU utilization threshold. If the average CPU load exceeds this, the script will increase the number of MPI processes.
- PULSE_TIME = 300: The interval (in seconds) at which the script checks the CPU utilization and potentially scales the number of processes. This determines how frequently the autoscaling logic is evaluated (currently set to 5 minutes).

Helper Functions

- run_command_nonblocking(command, shell=False): Executes a given command (as a string or list) in the background without waiting for it to complete. It returns the subprocess. Popen object, which represents the running process. This allows the script to start a process and continue with other tasks.
- get_process_output(process): Takes a subprocess.Popen object waits for the process to finish, and returns its standard output (stdout), standard error (stderr), and return code.
- is_process_running(pid): Checks if a process with the given process ID (pid) is currently running on the system using psutil.pid_exists().
- kill_process(process): Terminates a given process. It first tries a gentle terminate() and then forcefully kills it using kill() if it doesn't stop within a short time. It also prints messages indicating the process's status.
- run_process(process): Takes a subprocess.Popen object prints its PID and includes commented-out example code for potentially doing other work while the process runs or checking its status. It also includes a try...except...finally block to get the process output and ensure the process is killed in case of an error or completion.

autoscale_processes() Function

This is the core of the autoscaling logic.

- It initializes variables: command, process, pid to track the currently running MPI process, num_of_processes to the initial number of MPI processes (set to 8), and previous_cpu_avg to 0 to ensure the first iteration triggers an action.

- It enters an infinite while True loop, which represents the continuous monitoring and scaling process.

- Monitoring: Inside the loop, it first compiles (mpicc -g cpu-stats.c -o cpu-stats) and then runs the cpu-stats MPI program with the current num_of_processes. It captures the standard output, which is assumed to be the average CPU load across the MPI processes. Decision Making: It calculates the absolute difference between the current and previous average CPU load. If this difference is greater than MIN_THRESHOLD or if it's the first iteration (previous_cpu_avg == 0), it proceeds with scaling logic.

- Scaling Logic:

  - If the cpu_avg is below MIN_THRESHOLD (25%), it checks if a process is currently running (pid and is_process_running(pid)) and kills it. It then sets num_of_processes to 8 and starts a new MPI process running sample.c with this reduced number of processes.

  - If the cpu_avg is between MIN_THRESHOLD (25%) and MAX_THRESHOLD (75%), it similarly kills any running process, sets num_of_processes to 16 and starts a new MPI process with this increased number of processes.

  - If the cpu_avg is above MAX_THRESHOLD (75%), it kills the running process, sets num_of_processes to 32, and starts a new MPI process with this further increased number of processes.

- Throttling: After the scaling decision (or if no scaling was needed), the script pauses for PULSE_TIME (300 seconds) before checking the CPU load again.

- Updating History: Finally, it updates previous_cpu_avg with the current cpu_avg for the next iteration's change detection.

Remarks

Another approach may involves group of processes. We upscale my union of groups and downscale by interesction similar to set theory. At the nodes still has to be initialized at the start, therefore, those unused nodes that were already spawn result in wasted resources.

For future improvements, depending on the application's requirements, it might be beneficial to avoid terminating existing processes when scaling up the number of resources. For instance, if we currently have 4 processes running and the CPU load increases, we could scale up to 8 processes to better manage the workload without interrupting the initial 4. The following code snippets illustrate how to independently launch two sets of 4 processes:

```
mpicc -g sample.c -o sample && mpiexec -n 4 ./sample
mpicc -g sample.c -o sample && mpiexec -n 4 ./sample
```

However, when scaling down, we would need to explicitly terminate processes and maintain a record of the active process count to ensure accurate resource management.

8.4.2 Distributed Computing Patterns

Map-Reduce

Key-value must be immutable. A map function accepts a key and outputs a value. The next operation is grouping (intermediate reduce operation) user-specific reduce (same value have aggregated value) map-reduce is a functional programming paradigm as shown in Figure 8.1.

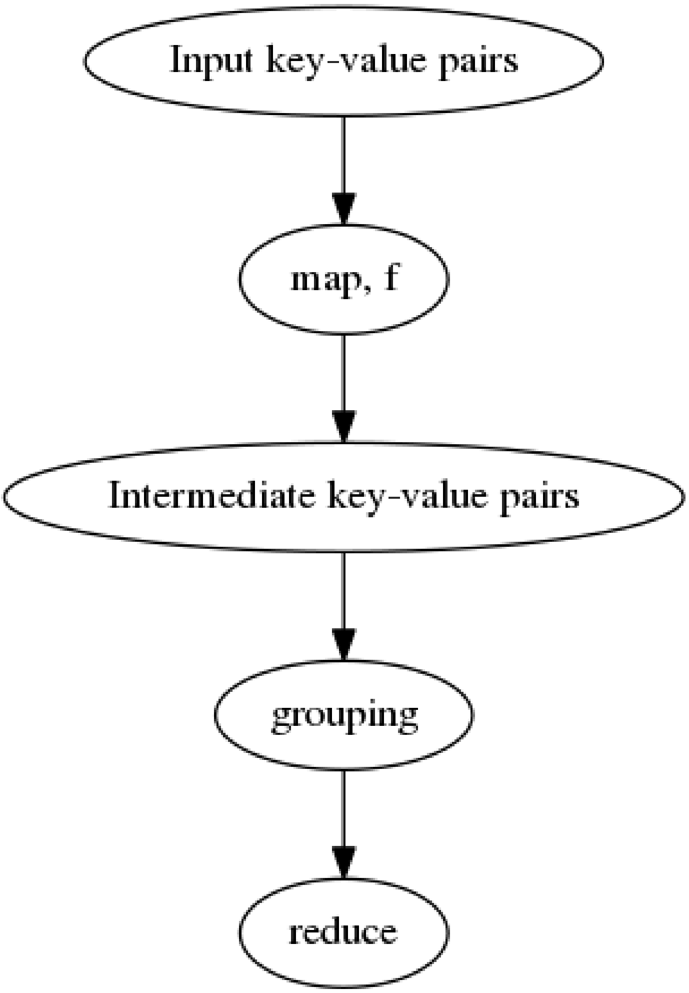


Figure 8.1: Map Reduce.

Spark key-value pair -resilient Distributed dataset (grouping) lazy evaluation (caching -reducing computation grouping) e.g td-idf, page-rank One possible way to approach it is to consider the problem in two cases Case 1: rank 0 has all data and every other node is empty

- split the data in chunks
- mpi-scatter to send to every process (or even mpi_send). An advanced user may send to a group of users to handle intermediate nodes where secondary reduction may be performed on a subset of nodes.

- Do computation between a subset of the node and set the result of the computation to a nonoverlapping set to aggregate the result.

Case 2: every node already.

- skip the mpi scatter phase in case 1.
- Regroup appropriately.

The author encourages the user to implement map-reduce using the description provided in this section.

Distributed Shared Primitive

The shared variable is protected with a lock to protect a critical section and compare and swap can be used to update the critical section in a non-blocking manner. I am not sure of how compare-and-swap works in mpi, my implementation of distributed shared primitive would be enhanced without blocking, which is more in line with our non-blocking approach in the core philosophy. if you adjust the resilience of your algorithm to enhance fault tolerance within the limit provided by quorum, then remember to cancel every outstanding request to prevent livelocks. All of our approaches to building a Distributed shared primitive are influenced by the work. One of our variations of a shared variable across processes made use of one-sided communication to pass messages between processes and always select the maximum. This works in the form of quasi-atomics and only works because our application makes use of the monotonic increasing property as an invariant. See an example of commented use in single-paxos3-snapshot.c.

```
int modify_var(struct mpi_counter_t *count, int valuein) {
    int *vals = (int *)malloc( count->size * sizeof(int) );
    int val;
    near_atomic_shared(count, valuein, vals);
    count->myval = MAX(count->myval, valuein);
    vals[count->rank] = count->myval;
    val = 0;
    for (int i=0; i<count->size; i++)
    {
        val = MAX(val, vals[i]);
    }
    free(vals);
    return val;
}
```

Another variation is just using the latest value as the total value shared across every process. This logic makes it easier to implement a compare and swap in the future if we decide to go fully non-blocking. See an example of use in single-paxos3-snapshot.c.

```
int reset_var(struct mpi_counter_t *count, int valuein) {
    int *vals = (int *)malloc( count->size * sizeof(int) );
    int val;
    near_atomic_shared(count, valuein, vals);
    count->myval = valuein;
    vals[count->rank] = count->myval;
    val = count->myval;
    free(vals);
    return val;
}
```

Distributed Hashmap

We have used the Lamport clock to make a distributed hashmap where every node has a local map (key-value pair) and in agreement, every node has the same key value key based on quorum.

8.5 Practical Considerations

- Who can partake in a leadership election? It is possible to designate nodes with permissions levels. One may decide if only nodes with write permissions or read permissions can partake in the election.

- Setting appropriate time drift to specific acceptable clock drift in your failure detectors.
- A hybrid approach may include a centralized server to allow late-joining clients to participate in future elections and get past information.
- Set range of rounds that a client can participate in an election
- The number of clients that can participate in quorum using the limits of the synchronous, asynchronous, and partially Synchronous.
- It is important to be mindful of failures and implement some form of bounded wait on the response and throw some timeout. It can also be useful for debugging failures. Livelock can be up in terminal conditions as a loop is not received to change to the terminal condition.

8.5.1 Tips for Testing

- Modularizing the source code and performing the unit test. Read more about the characteristics of concurrency bugs and inspire the tests to be written.
- Identify invariants as relations and use metamorphic testing to test with higher coverage.
- Add assert on bad conditions violations within the source code and scope within a condition flag that can be passed as a command-line argument to toggle the assertion when needed.
- Experiment on a testbed using a cluster of VMs in a LAN managed with a DevOps tool such as Vagrant, see hints.

8.5.2 Evaluation Metrics

- Message complexity
  - number of messages required to complete an operation
  - bit complexity (message length)

- Time complexity
  - communication steps
  - All communication steps take one unit. The time between send(m) and deliver(m) is at most one unit.

8.6 Exercises for the Readers

1. Extend the implementation of sequence Paxos to work beyond 4 processes as is the limit of our demonstration in sequence-paxos4.c using a shared distributed primitives. Take hints from single-paxos3-snapshot.c which work for any number of processes. The idea in our sequence Paxos implementation is to make the client send each item in an array one at a time, and once decided by the learner, send the index for the next item that the client has to send to the proposer.

2. Implement a three-phase commit as shown in SubSection 4.6.1. Use the source code implemented for two-phase commit (two-phase-commit.c) as a guide.

3. Write tests for the Distributed algorithm discussed in the book. We will give the user the task of implementing tests as an exercise.

4. Implement telemetry for experimentation on characteristics of any algorithm discussed in this book. Hint: see our design of communication of telemetry in single-paxos3-snapshot.c.

5. Set up a testbed with a simulated LAN with vagrant VM for running mpi cluster.

6. Implement network shaping using VM to test out different performances in varying network bandwidths.

7. Improve the autoscaling.py using our OpenMPI framework. See our implementation in SubSection 8.4.1, then roll out yours and enjoy the challenge. Hints: Use the ideas of incremental spawning of processes, rather than shutting every process down and restarting with the expected number of nodes.

8. This project involves training a fundamental linear regression model. The optimization method to be used is Stochastic Gradient Descent (SGD), a common algorithm detailed at Stochastic Gradient Descent. The overarching goal is to create the initial building blocks for a distributed learning system, specifically focusing on Federated Learning. Federated Learning is a technique that allows multiple devices or organizations to train a machine learning model collaboratively without sharing their data.

   To simulate this distributed environment, we will use the OpenMPI framework. Within this framework, each running process is an independent 'node' or participant in the federated learning system.

   For the data used in training, there are two possible approaches:

   - Decentralized Data Generation: Each individual node (MPI process) will generate its own set of randomized training data. This mimics a real-world federated setting where data is naturally distributed.
   - Centralized Generation and Distribution: The primary node (the process with rank 0) will create the entire training dataset. Subsequently, this central node will distribute portions of the data to all other participating nodes via broadcasting.

   This project emphasizes understanding the trade-offs of distributed learning, such as communication costs, data heterogeneity, and synchronization, with the primary goal of a basic, working federated linear regression implementation. The process involves each node training a local model on its data, followed by an aggregator collecting these pre-computed models to create a global model using model averaging (see model averaging).

   As a helpful starting point, consider the strategies for parallelizing linear algebra computations, as presented in Jeff Dean's bachelor thesis available at Jeff Dean's Bachelor thesis. These techniques for distributing and combining computations can offer valuable insights [1] [2] .

[1] For potential future development and expansion of this project, consider exploring the following ideas: including asynchronous late update and dropout

References

[1] BitTorrent (BTT) White Paper. https://www.bittorrent.com/btt/btt-docs/BitTorrent_(BTT)_White_Paper_v0.8.7_Feb_2019.pdf. Date accessed: September 5, 2022.

[2] FastTrack. https://en.wikipedia.org/wiki/FastTrack. Date accessed: September 5, 2022.

[3] Grid Computing. https://en.wikipedia.org/wiki/Grid_computing. Date accessed: September 5, 2022.

[4] KTH Distributed System Course. https://e-science.se/2020/05/course-on-reliable-distributed-systems-part-i/. Date accessed: September 5, 2022.

[5] List of CRDT resources. https://wiki.nikitavoloboev.xyz/distributed-systems/crdt. Date accessed: September 5, 2022.

[6] One-sided Communication in MPI. http://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture34.pdf. Date accessed: September 5, 2022.

[7] OpenMPI Examples. https://web.archive.org/web/20220305170744/https://hpc.llnl.gov/documentation/tutorials. Date accessed: September 5, 2022.

[8] OpenMPI Tutorials. https://github.com/mpitutorial/mpitutorial/tree/gh-pages/tutorials. Date accessed: September 5, 2022.

[9] Parallel, Concurrent, and Distributed Programming in Java Specialization. `https://www.coursera.org/specializations/pcdp`. Date accessed: September 5, 2022.

[10] Parallel Programming for Science and Engineering Using MPI, OpenMP, and the PETSc library. `https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/EijkhoutParallelProgramming.pdf`. Date accessed: September 5, 2022.

[11] Reliability Patterns. `https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency`. Date accessed: Novemberber 21, 2022.

[12] Summaries from MIT Distributed Systems Course. `https://people.csail.mit.edu/alinush/6.824-spring-2015/index.html`. Date accessed: September 5, 2022.

[13] Attiya, Hagit and Welch, Jennifer . *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley & Sons Inc., 2004.

[14] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. Pelican: A Building Block for Exascale Cold Data Storage. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 351–365, 2014.

[15] Beaver, Doug and Kumar, Sanjeev and Li, Harry C. and Sobel, Jason and Vajgel, Peter . Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[16] Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O'Shea, and Antony Rowstron. Virtual ring routing: Network routing inspired by dhts. *SIGCOMM Computer Communication Review*, 36(4):351–362, 2006.

[17] Miguel Castro, Manuel Costa, and Antony Rowstron. Debunking Some Myths

about Structured and Unstructured Overlays. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, pages 85–98, 2005.

[18] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[19] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies*, number 2009 in Lecture Notes in Computer Science, pages 46–66, 2001.

[20] Coulouris, George and Dollimore, Jean and Kindberg, Tim and Blair, Gordon . *Distributed Systems: Concepts and Design.* Addison-Wesley., 2012.

[21] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Tappan Morris, and Ion Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the Symposium on Operating Systems Principles, SOSP*, volume 35, pages 202–215, 2001. Operating System Review.

[22] David G. Andersen and Hari Balakrishnan and M. Frans Kaashoek and Robert Tappan Morris. The Case for Resilient Overlay Networks. In *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems*, pages 152–157. IEEE Computer Society, 2001.

[23] Allen Downey. *The Little Book of Semaphores.* 2016.

[24] Peter Druschel and Antony I. T. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Workshop on Hot Topics in Operating Systems, HotOS*, pages 75–80. IEEE Computer Society, 2001.

[25] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-Peer Membership Management for Gossip-Based Protocols. *IEEE Transactions on Computers*, 52:139–149, 2003.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of 19th ACM symposium on Operating systems principles*, pages 29–43, 2003.

[27] Brian Hall. *Beej's Guide to Network Programming Using Internet Sockets*. 2020.

[28] Brian Hall. *Beej's Guide to Unix IPC*. 2020.

[29] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8–45, 2007.

[30] Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *ACM SIGOPS Operating Systems Review*, 41(5):2–7, October 2007.

[31] Kubiatowicz, John and Bindel, David and Chen, Yan and Czerwinski, Steven E. and Eaton, Patrick R. and Geels, Dennis and Gummadi, Ramakrishna and Rhea, Sean C. and Weatherspoon, Hakim and Weimer, Westley and Wells, Chris and Zhao, Ben Y. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, 2000.

[32] João Leitão, José Pereira, and LuÍs Rodrigues. *Gossip-Based Broadcast*, pages 831–860. Springer, Boston, MA, 2010.

[33] Lynch, Nancy A. and Patt-Shamir, Boaz . *Distributed Algorithms: Lecture Notes for 6.852*. Course note for Graduate course in Distributed Algorithms, 1993.

[34] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. *Peer-To-Peer Systems: First International Workshop, IPTPS 2002, Cambridge, MA, USA*, 2002.

[35] Muralidhar, Subramanian and Lloyd, Wyatt and Roy, Sabyasachi and Hill, Cory and Lin, Ernest and Liu, Weiwen and Pan, Satadru and Shankar, Shiva and Sivakumar, Viswanath and Tang, Linpeng and Kumar, Sanjeev. F4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 383–398, 2014.

[36] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva P. Shankar, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic

Filesystem: Efficiency from Exascale. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, pages 217–231, 2021.

[37] Alex Petrov. *Database Internals: A Deep-dive into how distributed data systems works.* O'Reilly Media, Inc., 2019.

[38] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[39] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. pages 329–350, Berlin, Heidelberg, 2001. Springer.

[40] konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)* , pages 1–10, 2010.

[41] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolic. Solution: Mir-BFT: Scalable and robust BFT for decentralized networks. *Journal of Systems Research (JSys)*, 2(1):1–34, 2022.

[42] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[43] Tanenbaum, Andrew S. . *Distributed Systems: Principles and Paradigms.* Pearson Prentice Hall., 2007.

[44] Ian J. Taylor and Andrew B. Harrison. *Gnutella*, pages 181–196. Springer, London, 2009.

[45] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press., 2000.

[46] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal Verification of Practical MPI

Programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 261–270, 2009.

[47] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[48] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, pages 35–44, 2007.

[49] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.