Modern Distributed Systems
Version 0.10

Kishore Kothapalli and Smruti R. Sarangi

June 26, 2024

Contents

Preface

How to Use this Book

Acknowledgements

Thank a bunch of people and things here...

Chapter 1

Introduction

The study of distributed systems is multi-decade old with different flavors. The progress in distributed computing over the past few decades has led to a vast repertoire of techniques, models, systems, and programming frameworks. The seminal work of Lamport, Misra, OTHERS, laid the foundations on which one could reason about distributed systems and understand its myriad challenges.

The influence of distributed computing can be seen in many traditional and emerging areas of Computer Science such as programming languages, algorithms, operating systems, file systems, storage, group communication, Internet-of-Things (IoT), Blockchain systems, and so on.

In the last decade, there have been several developments with a huge impact on bringing computing to the benefit of the society. This book attempts to strike a balance between the foundational material related to the development of distributed computing and the principles behind the current billion-scale systems that are playing a key role in the growth of computing over the past two decades. These systems such as the Google File Systems, the Amazon DynamoDB, the Map-Reduce programming framework, and the like are also successful in bringing the theoretical developments in distributed computing to practice.

However, much of this progress has not yet made it to graduate level course text books that narrate and summarize the important design considerations behind these systems and the impact of the theory of distributed computing on the design choices. These details often serve to further the interactions between the communities and understand how theory and practice meet.

In the current scenario, the field of distributed systems has grown so

enormously that a single semester first level course will no longer be able to cover its entire scope. This leaves instructors needing to make suitable choices on whether the course will lean towards systems issues, algorithms, programming, or applications into emerging areas.

Existing textbooks also tend to lean more on one aspect of distributed computing. Another drawback of existing textbooks is their failure to cover many contemporary systems and topics. This results in very little knowledge about the system design space and choices, implementation details, user experiences, and the like. While there are established conference and journal papers that describe the above ideas, the scope of detail that can be provided in a textbook is more vast than a conference or a journal paper. In addition, a textbook style write-up gives more freedom to compare and contrast solutions while still presenting the core ideas in a simple yet comprehensive manner.

1.1 Types of Distributed Systems

Definition 1 *A distributed system comprises N independent processes that communicate with each other using a shared media such as shared memory or a message passing channel. Most algorithms and applications running on distributed systems make the entire system appear as a single process that can concurrently handle multiple user requests. Such systems that can typically scale automatically, handle faults, and hide details of the underlying hardware and software platforms.*

A distributed system comprises N independent processes – a process runs on a *node*, which can be a real machine or a virtual machine. . The processes in general act independently, unless specified otherwise. This is where we would like to define three models of computation [36]: synchronous, partially synchronous and asynchronous. In a *synchronous* system, the processes either share a clock or are loosely synchronized with each other. There is a known upper bound for the time it takes a message to get delivered (Δ) and the ratio between the speeds of two processors running the processes (η). Typically, we can divide the computation into a sequence of *rounds*. In each round all the processes perform a set of actions and all of them agree on the fact that the round has finished. Once all the actions in the previous round finish, a new round begins. Note that all points of time, processes are aware of the current round and the actions that they need to perform in this round.

In partial synchrony, Δ and η exist, but they are not known a priori. In other flavors of partial synchrony these bounds are known but they start holding from an unknown time T.

The more general setting for a distributed system is an asynchronous setting, where the processes do not share a physical clock and the algorithm cannot be broken into rounds – Δ and η are not defined. This is a purely event-driven system, where a process receives a set of messages and in response changes its state and sends some new messages. In a practical scenario, this setting is more useful because it does not relying on strict time synchronization.

As we shall see in Section **??**, most distributed systems find it very hard to deal with faults. Furthermore, assuming that a large system will not have faults is also not a practical assumption. We need to create protocols that continue to work even in the presence of faults.

Definition 2 *Let the upper bound for the time it takes a message to get delivered be Δ, and the maximum ratio between the speeds of two processors running the processes be η. In a synchronous system, Δ and η are defined and known. In a partially synchronous system, they are typically defined but not known. In some cases, they are known, but we don't know when they will begin to hold after an algorithm has started. In a purely asynchronous setting, these bounds don't exist.*

1.2 Fault Models in Distributed Systems

The main aim of any distributed system is to make the entire system appear as one. It can be thought of as a single machine that provides a bunch of services. In general, distributed systems are complicated and ensuring a good quality of service is hard. However, in most practical settings, we need to deal with node failures. The moment such faults in a system arise, the system can behave in very unpredictable ways. As a result, making a distributed system fault-tolerant is challenging, yet is required.

Let us distinguish between three terms that are often very confusing and in practice are used interchangeably: fault, error and failure.

Fault A *fault* is a deviation from ideal behavior in a system. Assume that there is a bug in the code. It is a fault.

Error Even if there is a bug in the code, we may not always be executing that piece of incorrectly written logic. However, when we execute the incorrect code, the internal state gets corrupted. This situation

represents an *error*. For instance, a node going down is an error. However, the entire system may not come down or the output may still be correct. An error is a manifestation of a fault but does not make the system unusable.

Failure A *failure* is defined as an event when the error is visible at the output and the system (to a large extent) is unusable. Either the output is wrong or it is not generated in the first place because the entire system goes down.

Distributed systems should be fault tolerant, which means that if there are parts of the system that are not designed correctly or not operating correctly, then the protocol should be designed to mask the effects of the fault. The fault should not progress to a failure. There are many ways in which a given node can fail in a distributed system. Note that a node failure may not imply a system failure, in fact in the scope of the overall system, a node failure may be a fault. Let us look at the different type of node failures.

Fail-stop The given node simply stops operating. This is visible to the rest of the system.

Fail-silent In this case, the failure is not easily visible to other components of the system. The system becomes unresponsive like the case of fail-stop failures. However, other parts of the system are not immediately aware of this fact. As a result, other processes may perceive the failed process to be just slow.

Byzantine failure This is the most pernicious type of failure. In this case the process can behave maliciously. It can either stop responding or provide erroneous outputs. It can also collude with other processes and try to confuse other honest processes or bring down the system. Such a process can appear failed to one process and appear perfectly functioning to another one. No assumption can be made about such processes.

Part I

Concepts

Chapter 2

Time in Distributed Systems

Segal's law is an adage that states: A man with a watch knows what time it is. A man with two watches is never sure.

Time and its measurement is a very fundamental physical concept. Ancient civilizations too practiced various ways of representing and measuring time with varying degrees of accuracy. People use the notion of time in many tasks including preparing their daily schedules, coordinating various events, ordering of events, and so on. It is important to note that in all of these applications, we have some inherent notion of how we benefit from (loosely) synchronized clocks. This loose synchronization suffices for most human tasks.

When we move to computer systems, one second of physical time corresponds to window in which a large number actions can be realized. The notion of time and causality therefore plays an important role in computer systems in general and distributed systems in particular. Within a single computer, time information is needed for the following use-cases, for example.

- Assigning Timestamps: For many events that take place in a computer, the operating system assigns a timestamp. Examples include the time at which a file is last touched or written to.

- Performance and Resource Usage: Timestamps offer a way for the system to monitor the performance of various applications and also track the usage of various resources by processes.

- Timeouts: Some user programs may need to stop after a certain time duration has elapsed. This requires the program to know the time.

- Scheduling and Statistics: The operating system may use time infromation in process scheduling. Further, the operating system may use time information to keep track of the system time assigned to each process, the overall time each process is running for, tracking login information of users, and other statistical information.

- Event Ordering: Timestamps are a way to arrive at a monotonic ordering of events that happen in the system.

From the above discussion, we note that one of the fundamental subsystems in a computer is the measurement of time. In the physical world, 1 second is measured as the time it takes the Cesium 133 atom to make exactly 9,192,631,770 transitions. Every computer needs to maintain the current time and use this current time in a varitey of situations. Most computers keep track of time by counting the number of oscillations of a quartz crystal. In particular, one second on a computer corresponds to 32,768 oscillations of a quartz crystal. However, this method is prone to a few errors arising due to the ambient state of the computer including heat. Therefore, unless left uncorrected, the time set and maintained by a computer can differ from the actual clock time. If that is the case with a standalone computer, maintaining time across a network of computers is usally much more challenging.

In the following, we first start with reviewing the solution space of algorithms for clock synchronization among computers on a network.

2.1 Solution Space

Having a synchronized clock in a distributed system is essential and the reasons cited earlier apply to such a setting also. Clock synchronization refers to the mechanisms used to align the clocks of nodes in a distributed system so as to enable applications mentioned earlier such as event ordering. There are primarily two mechanisms: synchronizing physical clocks, and using logical clocks. We briefly review the solution space of these two mechansims in this section.

2.1.1 Synchronizing Physical Clocks

Mechanisms for synchronizing physical clocks in distributed systems borrow from the same problem in the case of computers connected in a local network. It turns out that maintianing synchronized physical time across computers

on a network is usually difficult. Some of the difficulties arise from the fact that the local time at each computer can drift due to varying ambient conditions, network delays across communication links are unpredicatable and possibly unbounded, computers or communication links can fail, lack of easy access global knowledge which limit nodes to making decisions based on local knowledge, and other such related issues. These problems carry over, and are amplified in a distribtued system with additional challenges such as the lack of any centralized control and lack of limits on physical distances between the nodes.

This situation calls for the design of efficient algorithms and protocols that address how computers on a network can synchronize their clocks. Starting with the early works of Marzello [**?**] and Ellis [**?**], there have been a sequence of developments in this direction.

Algorithms for this problem use a range of techniques including peer-to-peer exchange, and master-slave models for synchronizing time across computers in a network. The latter case is easier to solve since computers on the network have a centralized server. The quintessential technique, in both the cases, is for a pair of computers to estimate the average transmission delay and their clock offset by using a sequence of ping and reply messages. Algorithms that assume an upper bound on the transmission delay are usually simpler than those that do not make such assumptions.

Other considerations that play a role in the design of these algorithms include the size and extent/spread of the network, the desired accuracy of the synchronization, and the auxiliary support available. There are theoretical limits on the accuracy acheived by *any* algorithm. In particular, Lundelius and Lynch [**?**] show that of n clocks cannot be synchronized with synchronization error less than $(\max - \min) \cdot (1 - \frac{1}{n})$, where max and min denote the maximum and minimum transmission delays across the computers in the network. Practical limits on the accuracy depend on multiple factors including the network conditions, the accuracy of the clocks at each computer, and the like.

Most of the algorithms for time synchronization employ a range of techniques for fault-tolerance. For instance, in a ping and reply interaction, if one side does not respond within a reasonable time period, the other side will typically call this round of interaction as unsuccessful and ignores the measurements from such unsuccessful rounds. Given the uncertainities in message transmission and other potential pitfalls such as broken or misbehaving clocks, another common technique is to discard also measurements that significantly deviate compared to worst-case scenarios.

2.1.2 Logical Clocks

Another direction that distributed systems use to measure time is to do away with synchronizing physical time and use logical time. Lamport [**?**] proposed such a logical time framework and the ability of logical time to solve many of the problems in distributed systems such as event ordering, mutual exclusion, using only logical time instead of physical time. One key assumption in the logical clock framework is that all communication happens within the system and there are no other communication channels.

2.1.3 Recent Developments

Usecases such as distributed transactions, financial transactions, and distributed databases have rekindled interest in synchronizing time in a distributed system. For such applications, and for applications that need the relationship of events with respect to physical clock, logical clocks do not suffice. Further, such applications require precision that is of the order of nanoseconds [**?**]. These usecases led to the design of physical clock synchronization algorithms that used also novel techniques from signal processing and estimation to achieve better accuracy.

In addition, current planet-scale distributed systems have multiple channels of communication that negate the assumptions of logical clock frameworks. These developments necessitated the design of clock synchronization mechanisms that use a combination of physical clocks and logical clocks. Recent attempts include hybrid logical clocks that use a combination of physical clocks and logical clocks, and providing for synchronized physical time at planet scale networks.

Most large organizations of today that deal with planet scale distributed systems use custom procotols for time synchronization by relying on technologies such as atomic clocks, dedicated and fault-tolerant communication infrastructure, GPS systems, and the like. In this direction, we describe the Google TrueTime TrueTime protocol that aims to emulate the solution on a network to the distributed system scale.

In the following, we provide sample points of solutions from the above mentioned space of solutions. We review how a computer on a network sets and maintains its time. This is followed by looking at the limitations of the solution designed in the context of a network to a large scale distributed system. We then move on to how time is maintained in a distributed system using logical clocks.

Finally, we discuss a solution that combines aspects of logical time and

physical time, Hybrid Logical Clocks (HLC), to arrive at a simpler way of maintaining time in a distributed system.

2.2 The TEMPO Protocol

One of the early algorithms/protocols to maintain the time in a local network is the TEMPO protocol of Gusella and Zatti [**?**]. This algorithm adjusts the time of the day at devices in a local network. Based on this algorithm, the Berkely UNIX 4.2 BSD also introduced a system call `adjtime()` that allowed for setting the time on a system.

2.2.1 The Algorithm

Consider the situation that process A and process B running on two different machines on the network want to estimate the clock skew between the machines. Process A sends a message to process B. On receiving such a message, process B sends a reply to process A with a timestamped message. Before sending the message, process B computes the time taken for the message from A to reach B as:

$$d_{A \to B} = \text{timestamp}_B - \text{timestamp}_A \tag{2.1}$$

Notice however that the clocks as A and B may themselves be in error by quantities e_A and e_B, respectively. In other words, $\text{timestamp}_A = t_A - e_A$ and $\text{timestamp}_B = t_B - e_B$. Using the above two, and denoting the trasmission delay from A to B as D_{AB}, we rewrite in Equation 2.1 as:

$$d_{A \to B} = t_{B_1} - e_{B_1} - t_{A_1} + e_{A_1} = D_{AB} + \delta - \text{Error}_{AB_1} \tag{2.2}$$

In Equation 2.2, the quantity D_{AB} represents the trasmission delay from A to B and δ refers to the offset we are trying to estimate.

As B sends a reply to process A, the above calculations can be repeated at process A to obtain the following. We denote the trasmission delay from B to A as D_{BA}.

$$d_{B \to A} = (t_{A_2} - t_{B_2}) - (e_{A_2} - e_{B_2}) = D_{BA} - \delta = \text{Error}_{AB_2} \tag{2.3}$$

Using Equations 2.2, 2.3, process A can also compute the difference in the transmission delay between A to B and B to A as:

$$\Delta' = \frac{d_{AB} - d_{BA}}{2} = \delta + \frac{D_{AB} - D_{BA}}{2} - \frac{\text{Error}_{AB_1} - \text{Error}_{AB_2}}{2} \tag{2.4}$$

We now see how we can repeat the above steps to estimate δ. Assume that the random variables corresponding to Error_{AB_1} and Error_{AB_2} are independent and symmetric. We can repeat the above experiment N times so that the average Δ' can be computed as follows.

$$\begin{aligned} \Delta' &= \frac{\sum_{i=1}^{N} \frac{d_{AB} - d_{BA}}{2}}{N} \\ &= \delta + \frac{\sum_{i=1}^{N} \frac{D_{AB} - D_{BA}}{2}}{N} - \frac{\sum_{i=1}^{N} \frac{\text{Error}_{AB_1} - \text{Error}_{AB_2}}{2}}{N} \end{aligned} \tag{2.5}$$

Notice from Equation 2.5 that the second and the third terms in the right hand side have a mean of 0. Hence, for large N, by appealing to the Strong Law of Large Numbers, the right hand side converges to δ.

This observation can now be used to design the protocol that computers on a local network use to synchronize their clocks. The TEMPO protocol designates one computer in a local network as a *master* and the rest as *slaves*. The master is responsible for initiating and coordinating time synchronization. The master uses the following steps.

1. The master interacts with each of the slave machines, say $S_1, S_2, \cdots$, and obtains estimates of $\Delta'_{S_1}, \Delta'_{S_2}, \cdots$, with respect to each of the slave machines.

2. The master then computes the average of the quantities, $\Delta'_{S_1}, \Delta'_{S_2}, \cdots,$, as the network average error.

3. The master directs slave machine S_i to adjust its clocks by an offset equal to the difference of Δ'_{S_i} and the network average error.

Notice that the above protocol requires some slave machines to set their time to be in the past. This can create situations that are not consistent.

2.2.2 Fault Tolerance

The TEMPO protocol is designed with simple fault tolerance. The failure of a slave to communicate with the master wihtin a specified timeout interval lets the master conclude that the slave machine non-operational. Similarly, if the slave machines do not recieve any message from the master within a specified time, they conclude that the master machine is unavailable and start the procedure to elect a new master.

2.3 The Network Time Protocol (NTP)

The Network Time Protocol (NTP) is described as RFC 958 [91]. This protocol indicates how computers on a network should periodically set their time so that all computers on a network have a consistent notion of the physical clock time. TRUE? To this end, the protocol organizes computers into two sets of entities such as Clients and Servers . Clients seek time from a server within the same network and are thus passive in the protocol. Servers are arranged into multiple strata and servers within a strata are called as *peers*.

Stratum 0, the highest level of strata, consists of the most accurate set of time servers, these are also called as *reference clocks*. These use devices such as GPS clocks, atomic clocks, and the like and are highly accurate.

Devices in Stratum 1 have a direct connection to one or more devices in stratum 0 in addition to having connections to their peers in the same stratum. Devices in this stratum are also known as primary clocks or primary time servers. They might be few microseconds in offset compared to the time in Stratum 0 clocks. These primary servers can also work as fallback clocks to sync up with each other just in case stratum 0 servers are unavailable.

Devices in Strata 2 and 3 synchronize with devices in the next higher strata and with those in the same strata. Figure 2.1 shows a typical set up of servers into strata.

There are two types of communication between the servers to obtain the current time: one set of communications between the peers, and one from devices in one strata to those in a higher strata. Since servers across strata and also those within a strata may have differences in clocks, and these differences can grow arbitrarily over time, an initial set of synchronization may not be enough. Therefore, peers in the NTP protocol exchange periodic messages to (re)set their local time. The interaction between two peers A and B is characterized by the following messages.

Peer A sends a message to peer B and records the time that the message is sent as T_1. Peer B records the time at which the message is recieved at B as T_2 and sends an acknowledgement to peer A at time T_3. This acknowledgement is received at peer A at time T_4. Each such message and acknowledgement is called as a trial . Figure 2.2 shows these four messages across A and B.

For each trial, the clock offset θ and the roundtrip delay δ of peer B relative to peer A at time T_4 are approximately given by the following. Measured at peer B, $T_2 = T1 + \delta/2 + \theta$, and $T_3 = T4–\delta/2 + \theta$. Add the

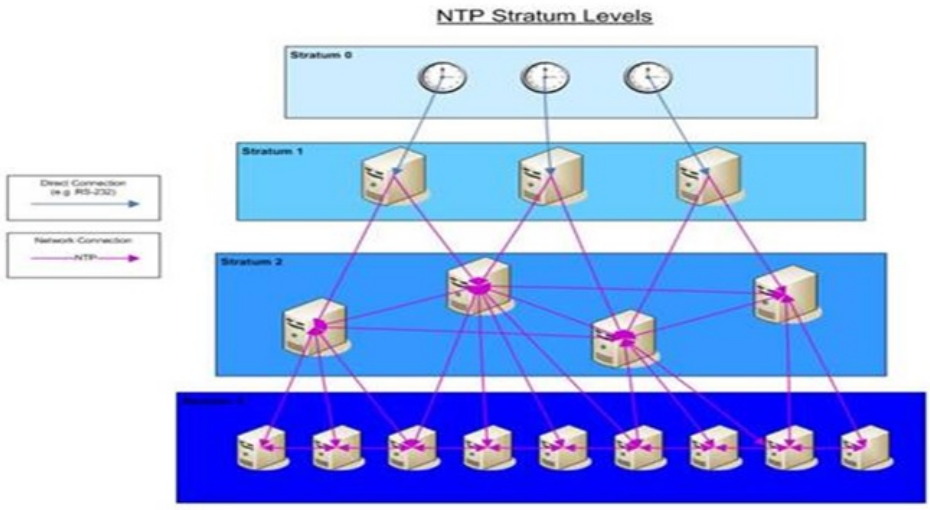


Figure 2.1: A set of servers arranged into various strata in the NTP protocol. Figure borrowed from ntpserver.wordpress.com .

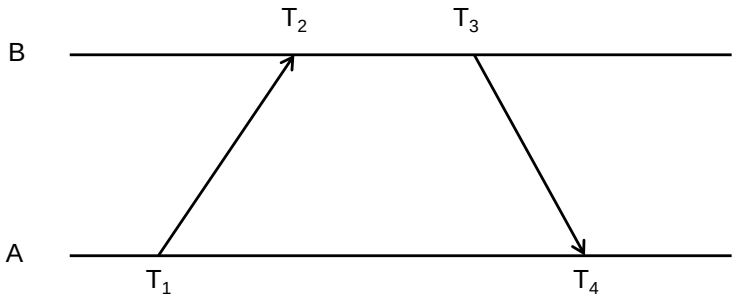


Figure 2.2: A pair of messages exchanged by A and B with send and receive times at A and B recorded at respective machines.

two equations to get $2\theta = a + b$ with $a =$ and $b =$. The round-trip dealy $\delta = T_3 - T_1 + T_2 - T_4 = a - b$.

Each peer maintains pairs (O_i, D_i), where O_i is a measure of the offset (θ) and D_i is a measure of the transmission delay of two messages (δ). The eight most recent pairs of (O_i, D_i) are retained. The value of O_i that corresponds to the minimum D_i is chosen The offset corresponding to the minimum delay is chosen as the offset O.

2.3.1 Limitations of NTP

Notice from the above that by having a set of continuously interacting peers, it is possible to have a physical time that is synchronized across computers. The typical errors in synchronization are small. However, in a large scale distributed system, the typical errors can go up to hundreds of milliseconds.

NTP works on the premise that all the systems can be arranged in a hierarchy and the required peering support can be set up. This assumption is difficult to maintain in a fully distributed system. Even if such a hierarchy and peering support is created, the typical physical distances can potentially involve large round trip times resulting in reduced accuracy of the agreed time. In particular, the accuracy using NTP in networked systems is of the order of few milliseconds. This error can increase to a few hundreds of milliseconds in rare cases. (See the experiment suggested in Question ???). This accuracy compares very infavourably to running NTP on a LAN which can produce accuracy to the order of less than a millisecond. A global scale distributed system may not be able to function correctly using physical time that has errors of the order of milliseconds or hundreds of milliseconds. These reasons indicate that NTP in its present form is not usable in distributed systems.

Nevertheless, distributed systems too need to measure time for a variety of reasons. Some of these are listed below.

- Distributed Transaction Processing: For instance, measuring time and assigning timestamps is important in transaction processing.
- Distributed Algorithms: Timestamps may also be essential in applications such as mutual exclusion. In some situations, dstributed systems use the notion of timeout to abort actions. This requires measuring elapsed time.
- Tracking Dependencies Among Events:
-

Given the above requirements and the difficulty of adopting the NTP approach, what do distributed systems do to maintain time? It turns out that by their nature, distributed systems make progress in spurts and one way to measure time is to understand how various events in a distributed system can be ordered at least in a partial order. This partial order, also known as causality, will then serve as a way of measuring time.

The solutions that distributed systems adopt vary from not using physical time, creating a robust distributed scale NTP-like framework, and using a combination of these two approaches. In the following, we will describe these three approaches in that order.

2.4 Logical Time

In an seminar paper, Lamport [78] showed that a partial ordering of events in a distributed system can provide a mechanism to set the notion of time and its measurement in a distributed system. Such a mechanism allows distirbuted systems use what is known as logical time instead of physical time.

Logical time is a mapping from events in a distributed system to a time domain. The events in a distributed system correspond to local computation, message send, and message receive. The distributed system in action is thus a system of computers processing events. Events within a machine and those across machines do have causal dependencies of the following kind. An event corresponding to a message send at a machine P has to happen *before* the message is received at machine Q. These causal dependencies induce a partial order on the set of events across machines in a distributed system.

Logical time aims to capture the partial dependencies between events. To make the description formal, let us denote by E_i the set of events happening at processor P_i for $i = 1, 2, \cdots,$. Let $E := \cup E_i$ denote the entire set of events happening at all processors. Let us also number the events that happen at P_i for $i \geq 1$ such that an event e_i^x occurs before e_i^y if $x < y$, for positive integers x and y. In other words, we say that at P_i event e_i^x happens before event e_i^y. Similarly, if P_i and P_j are two differerent processors, and P_i sends a message to P_j, the corresponding send event must happen before the receive event. We used $\rightarrow_{msg}$ to denote this relation. Finally, we use the binary relation $\rightarrow$ across pairs of events and is defined as follows. Consider any two events

e_i^x and e_j^y happening at processors i and j.

$$e_i^x \rightarrow e_j^y \iff \begin{cases} i = j \text{ and } x < y, & \text{or} \\ e_i^x \rightarrow_{msg} e_j^y, & \text{or} \\ \exists e_k^z \in E \text{ such that } e_i^x \rightarrow e_k^z \text{ and } e_k^z \rightarrow e_j^y \end{cases}$$

Notice from the above definition that indeed the relation $\rightarrow$ is a partial order. There could be events that are not related according to $\rightarrow$. We say that for two events e_1 and e_2, if $e_1 \rightarrow e_2$ or viceversa then the two events causally affect each other. Otherwise, the two events are said to *logically concurrent*. We use $e_1 \parallel e_2$ to indicate that the two events are logically concurrent.

We now build on the relation $\rightarrow$ between pairs of events to define the notion of logical time. Just like physical time, logical time also assigns a timestamp to each event. These timestamps ensure that if an event e_1 that causally affects another event e_2, then the timestamp of e_1 is smaller than that of e_2. One can infer the causality beween events by comparing the (logical) timestamps.

A *logical clock* $\mathcal{LC}$ is a function that maps an event e in a distributed system to an element in the time domain T, denoted as $\mathcal{LC}(e)$ and called the timestamp of e, and is defined as a mapping $\mathcal{LC} : E \rightarrow T$ such that the following property is satisfied. For two events e_i and e_j, $e_i \rightarrow e_j \Rightarrow \mathcal{LC}(e_i) < \mathcal{LC}(e_j)$. This monotonicity property is called as the clock consistency property of the logical clock system.

Based on the above idea, it is possible to define several systems of logical time. These systems have the following commonalities. To use the timesystem, each processor in the distributed system has two data structures. A *local logical clock* denoted lc_i at processor P_i helps P_i measure its own progress. A *logical global clock* denoted gc_i at processor P_i is a way of measuring the view of the logical global time of process P_i. The secod of these is required so that P_i can assign consistent timestamps to events that occur at P_i. The logical time system also provides for rules to update the data structures so as to ensure the time consistency condition. In the following, we study the logical scalar time system proposed by Lamport [78].

2.4.1 Logical Scalar Time

In this model, a combined integer c_i represents the data structures lc_i and gc_i at process P_I. The time domain is the set of non-negative integers. There is a common increment d by which the time is incremented as necessary.

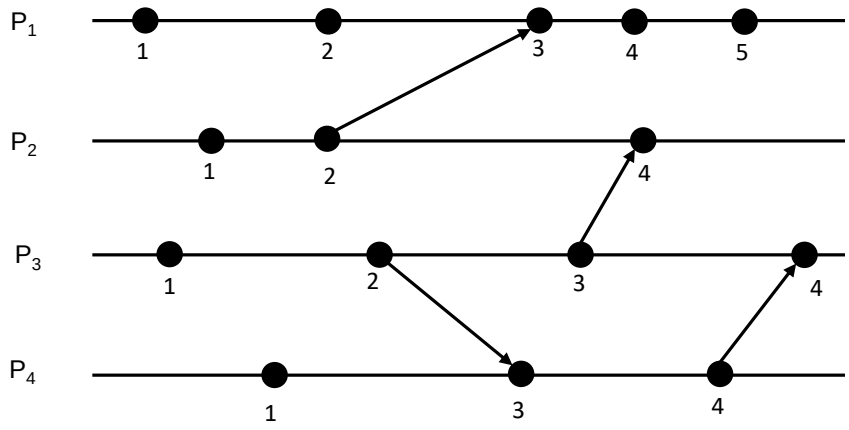


Figure 2.3: A set of four processors using scalar time.

Process P_i, before executing an event e increments its local time c_i to $c_i + d$. Process P_i also includes its current time on every message that it sends. This piggybacking of the current time, denoted c_{msg}, on each message helps the recipient P_j to update its time as $c_j = \max\{c_j, c_{\text{msg}}\}$. Subsequently, P_j increments its local time c_j by d before delivering the message. These steps ensure that the timestamp of the corresponding message receive event is larger than the timestamp of the message send event. Figure 2.3 shows an example.

It can be noticed that the scalar time as described ensures the consistency property. Extending the notion of consistency, a logical time is said to be strongly consistent if it holds that for any two events e_i and e_j, $e_i \rightarrow e_j \iff \mathcal{LC}(e_i) < \mathcal{LC}(e_j)$. Logical scalar time is however not strongly consistent. Check Figure 2.3 and the events with timestamp 5 and 4 at processes P_1 and P_2, respectively. While $\mathcal{LC}(e_i) < \mathcal{LC}(e_j)$, it does not hold that $e_i \rightarrow e_j$.

Logical scalar time has other properties. If we set the increment $d = 1$ at all times, then an event e wkith a timestamp of t depends on exactly $t-1$ events in the past. In addition, the scalar time can be used to induce an ordering on events as follows. Consider the tuple (t, i) associated with every event e so that t is the scalar timestamp of e and i is the id of the processor where the event e happens. For two tuples (t_1, i_1) and (t_2, i_2), we define $(t_1, i_1) < (t_2, i_2)$ iff $t_1 < t_2$ or if $t_1 = t_2$ and $i_1 < i_2$. Under this ordering, it holds thatfor any two events e_1 and e_2, $e_1 < e_2 \rightarrow$, either $e_1 \rightarrow e_2$ or $e_1 \parallel e_2$.

The notion of logical scalar time has found applications in several fun-

damental problems in distributed computing. For instance, we will see in Chapter 4 that scalar time is useful in designing an algorithm for ensuring mutual exclusion in the distributed setting.

To address the limitation of scalar time that it does not satisfy strong consistency, other logical time systems include the vector time[1] [40, 82, 88] and its extension to matrix time [118]. We breifly review the notion of vector time now.

2.4.2 Logical Vector Time

Logical scalar time uses only one variable to represent both the local logical time at a processor and also the logical global time. Vector time solves this issue by representing time as a vector of n dimensions where n is the number of processors in the distributed system. Let $V_i[1..n]$ denote the vector at process P_i. Each element of the vector is a non-negative integer. The ith component of the vector V_i is the local logical time at processor P_i. In addition, the jth component of V_i represents the latest knowledge of P_i about the local time at process P_j. In particular, if $V_i[j] = x$, then process P_i knows that local time at process P_j has progressed till x. The entire vector V_i constitutes the view of the global logical time at P_i and is used by P_i to timestamp events. The timestamp of an event is now a vector of n dimensions.

When a process P_i executes an event e, it increments $V_i[i]$ by d and assigns e the timestamp of V_i. When a process P_i sends a message to a process P_j, the current time V_i is sent along with the message as V_{msg}. On receipt if the message, process P_j updates V_j as follows. For each $1 \leq k \leq n$, $V_j[k] := \max\{V_j[k], V_{msg}[k]$. This is followed by incrementing $V_j[j]$ by d before delivering the message. This ensures that the event corresponding to the receipt of the message has a larger timestamp than that of the corresponding send event. Figure 2.4 shows an example of using vector time.

To compare two vector time stamps, we use the following rule. Let v and w be two vectors of n dimensions each. We say that $v = w$ if for every $1 \leq i \leq n$, $v[i] = w[i]$. We say that $v \leq w$ if for every $1 \leq i \leq n$, $v[i] \leq w[i]$. We say that $v < w$ if $v \leq w$ and there exists an index i $(1 \leq i \leq n)$ such that $v[i] < w[i]$. Finally, we say that $v \parallel w$ if neither $v < w$ nor $w < v$. As an example, the vector $[1, 3, 4]$ is less than the vector $[1, 5, 6]$. The vectors $[2, 5, 3]$ and $[3, 4, 4]$ are concurrent.

[1]It has long been not clear as to who is the first one to propose vector time. See the blog by Kuper `https://decomposition.al/blog/2023/04/08/who-invented-vector-clocks/`

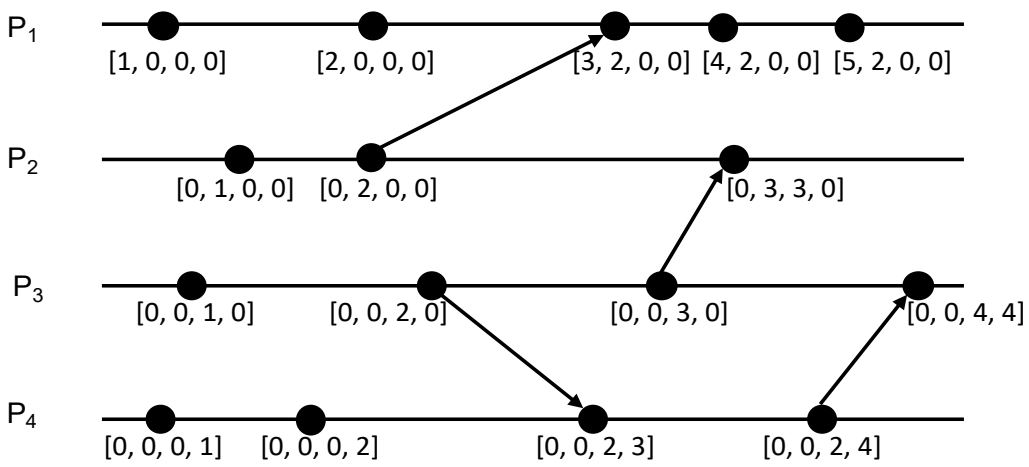


Figure 2.4: A set of four processors using vector logical time.

From the definition of vector time, it is apparent that the ith component of the vector clock at process P_i, $V_i[i]$, denotes the number of events that have occurred at P_i until that instant. If an event e has timestamp v, then $v[j]$ denotes the number of events executed by process P_j that causally precede e. Further, $(\sum_{j=1}^{n} V[j]) - 1$ is the total number of events that causally precede e in the distributed computation.

It can be noted that vector time is strongly consistent. For any two events e and f, it is possible to determine if the events are causally related.

Optimized Implementation of Vector Time

In both of these extensions, the data strucutres used for representing time increase with the increase in the number of systems. This can pose a limitation trouble especially since the current time is sent on each message. There are some techniques to reduce the overhead of the time information carried on each message. One such technique, due to Sinhgal and Kshemkalyani [107] is to use two additional information at every process in the form of two arrays. The LastUpdate array corresponds to when each process i last updated its information about the time at a process j, and the LastSent array corresponds to the time when process i last sent its time information to process j. With the help of these two arrays, when process i needs to piggyback a message with its current vector clock to process j, it includes only entries from its vector clock, $vt_i[k]$ such that $\text{LastSent}_i[j] < \text{LastUpdate}_i[k]$. However, these techniques are not generic and work only in a heuristic manner.

We refer the reader to [107] for more details on these optimzations.

There are other mechanisms that optimize the amount of time information to be sent as piggyback on messages. Notable ones include the direct dependency work of Fowler and Zwaenepoel [44] and that of Jard and Jourdan [65]. In the direct dependency technique of Fowler and Zwaenepoel [44], processes maintain information pertaining to direct dependence of events on other processes. The actual vector time of an event can be computed offline by using a recursive search on the recorded direct dependencies to account for transitive dependencies. In particular, if an event e_j at a process P_j occurs before the event e_i at process P_i, it can be inferred that events e_0 to e_{j-1} at process P_j also happen before event e_i. In this case, it suffices to record at P_i the lastest of the events at P_j the happened before the event e_i. In a similar fashion, P_j would record the latest event that e_j would directly depend on. This recursive trail of dependencies can be used to obtain the vector clock of e_i when needed. One important step in implementing this approach is to make sure that every process updates the dependency information after receiving a message and before it sends out any messages.

More details and the recursive algorithm needed to identify the transitive dependencies are in [44]. Since this technique involves additional computation, its applicability is limited to specific settings such as causal breakpoints and asychbronous checkpoint recovery.

Beyond vector time, Wu and Bernstein [118] shows use cases for matrix time. In matrix time, each processor stores time as a two dimensional matrix. At processor P_i, the (i, j)th entry in the matrix time represents the progress at Processor j that P_i is aware of. At processor P_i, the entry at (k, ℓ), for $k \neq i$, refers to the progress that processor P_i knows about what processor P_k knows of the progress at P_ℓ. While being useful in some specific applications, matrix time has the big disadvantages including the size of the timestamp to be sent along every message.

2.4.3 Limitations of Logical Time

The solution proposed by Lamport [78] to circumvent using physical time in a distributed system and instead use logical time is elegant. However, logical time has its own shortcoming that impact its practicality. If one insists on strong consistency, then scalar clocks do not suffice and one needs to use vector clocks or matrix clocks. These come with a huge overhead on message sizes and the overhead grows with increasing size of the distributed system. Recall from Section 2.4.2 that the timestamp information in a system using vector clocks grows linear in the number of nodes in the system. With

current emphasis on large-scale systems, such an overhead is not practical.

In addition, even as logical time can be used to provide for a total order of the events there is no gurantee that two events e_1 and e_2 with the timestamp of e_1 smaller than that of e_2 are such that e_1 indeed occurs physcially before e_2. Further, the total order induced is not unique. Such inference may be needed in distributed systems especially in cases where one wishes to enforce certain notions of consistency. Examples include mutually exclusive access to resources which is possible to solve using the total ordering induced by logical clocks but the solution does not ensure that the access is granted in physical time order.

Logical clocks also assume that all interactions of the participants in the distributed system happen within the system and there are no other communication channels. With the rapid rise of cyber-physical systems and Internet-of-Things, there is now more scope for systems to talk via multiple channels and today's systems are highly integrated in their operational environment but yet loosely coupled.

From the preceeding discussion, we note that the existing solutions fall short of the practical requirements. Using physical time comes with limitations as mentioned by Lamport [78]. In fact, the solution from [78] makes two assumptions. Firstly, the clock C_i at any process i can only be set forward and never be set back. Secondly, the network of processes is to be strongly connected. Thirdly, it is also assumed that there exist two constants δ and p such that each link carries a message whose delay is unpredictable yet bounded by δ every p seconds. Finally, the offset between the clocks at any two processes in the system is bounded by $d(2\kappa\delta + p)$ where κ is a constant that bounds the drift in the clock at any process over time, and d is the diameter of the network.

2.5 NTP at Distributed Scale

Corbett et al. [29] showcases a few details of how to engineer and run a protocol similar to NTP in a modern planet-scale distributed system. A brief summary of TrueTime follows.

Consider a setting where datacenters are spread across the planet. Each datacenter has one or more TimeMaster servers, also known as time servers. The TrueTime solution has two kinds of TimeMaster servers.

- GPS Time Master: These TimeMaster servers contain GPS receiver nodes and can interface with GPS signals and receive time infromation via satellites. Figure 2.6 shows an example TimeMaster set up.

- Armageddon Master: These TimeMaster servers contain atomic clocks that are highly accurate. Time information from atomic clocks acts as a supplement to the time information from the GPS TimeMasters in case the satellite connections suffer any unexpected outage.

The engineering choice of using GPS based time information and atomic clocks is to account for good redundancy. Usually, it is observed that the factors that affect the failure of these two time systems are independent. For instance, satellite communication on which GPS relies on may be impacted by interference from radio waves. These radio waves do not impact the functioning of atomic clocks.

TimeMasters periodically exchange and compare their time with other TimeMasters. This exchange is along the lines of the NTP protocol. Each TimeMaster also checks the local clock for integrity. In case of these checks fail, the TimeMaster voluntarily stops being a TimeServer pending resolution of the failure.

We now outline how clients use TrueTime. A client pings a set of TimeMaster servers of both kinds, the GPS TimeMasters and the Armageddon TimeMasters, across the network. As in NTP, from the replies received from the TimeMasters, clients use statistical techniques to drop the outlier responses and know the best candidate time from the set of responses. While these actions are going on, clients advertise a time interval that has increasing degree of uncertainty. This uncertainty is in proportion to the local clock drift.

Applications using TrueTime follow the TrueTime API. In this API, the timestamp of an event is an interval that accounts for the limited uncertainty. Each timestamp has the datatype TTInterval of form [earliest, latest] which correspond to the beginning and ending times of the interval. The API has three functions: the function TT.now() which returns the current timestamp as the above interval, the function TT.after(t) which returns True if time t has certainly elapsed, and the function TT.before(t) which returns True if time t is certainly yet to occur.

TrueTime guarantees that for an invocation of $tt = TT.now()$, $tt.earliest \leq t_{abs}(e) \leq tt.latest$, where is the absolute time when event happened. The interval given by *earliest* and *latest* are guaranteed to be within a bounded uncertainty. In practice, Google states that this interval lasts between 1 ms to 7 ms.

To summarize, notice that TrueTime brings two novel contributions. One is to represent time as an interval instead of as an absolute scalar quantity. The second novel aspect of TrueTime is to utilize GPS and atomic

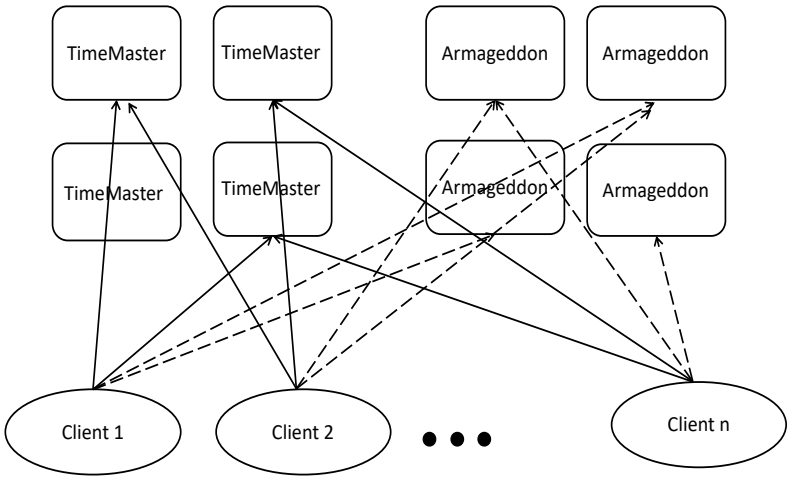


Figure 2.5: An illustration of the Google TrueTime time architecture. Solid lines represent clients interacting with TrueTime Master and dashed lines incidate client interacting with Armegeddon master.

clock based time references. The latter allows TrueTime to use algorithms similar to that of NTP to synchronize time and provide strong guarantees on the interval of uncertainty.

TrueTime faciliated many distributed applications that rely on the availability of time. A good example is the Google spanner distributed database system that uses TrueTime to induce a strict concurrency control for transactions at a global scale. More details of this is provided in Chapter 9.

2.5.1 Other Related Systems

The design of TrueTime by Corbett et al. [29] led to the desing of similar systems. Amazon launched its version of time service labled as timesync using GPS based and atomic clocks. This service also uses many ideas similar to that of TrueTime and NTP. The TimeSync service is now used in similar Amazon products such as Amazon DynamoDB [108].

Facebook in 2020 also moved to a distributed scale NTP engineering effort called `Chrony`. Details of this can be seen in [38].

2.5.2 Limitations of TrueTime and Similar Systems

Truetime and other related systems come under a class of highly engineered solutions that require the presence of speical purpose hardware on time

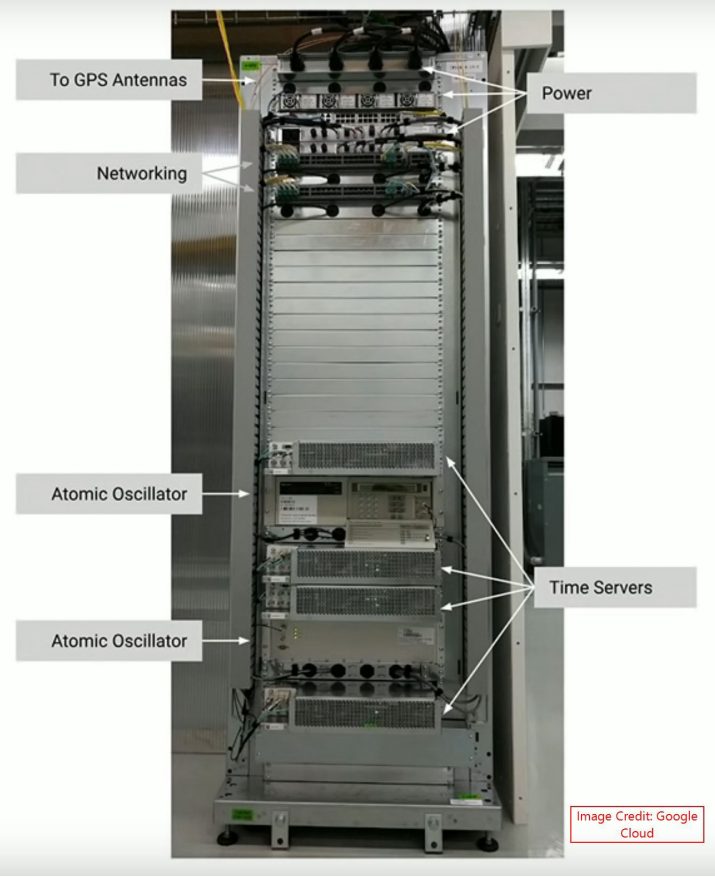


Figure 2.6: An illustration of the Google TrueTime time master. Picture borrowed from Google Cloud.

servers in the form of GPS receivers and precision atomic clocks. The atomic clocks used by Google are accurate to the order of a difference of 1 second in 10 million years. Such high level of precision instrumentation cannot be feasible for many settings. Secondly, it is not the hardware and instrumention alone that supports systems such as TrueTime. The communication latency in the interactions between clients and TrueTime Master servers and the Armageddon Masters can play a role too. In particular, the TrueTime system relies on having a dedicated, fault-tolearant, high-speed communication fabric that ensures that the window of uncertainty in the TrueTime system is as small as possible.

In addition, TrueTime returns a time interval as a timestamp. When the interval is small enough, this sufifces to order events and compare the timestamps of events to know which event preceeds another event. In the unlikely situation that such an ordering is not possible, TrueTime recommends that applications wait out the period of uncertainty. These introduce unexpected delays in application event processing and also reduced the concurrency achievable in the system.

2.6 Hybrid Logical Clocks – Combining Physical and Logical Times

Kulkarni et al. [76] combine ideas from logical time and physical time to arrive at a simpler way of maintaining time in a distributed system. They call this as Hybrid Logical Clocks (HLC). In the following, we summarize the basic ideas from [76].

The goal of HLC is to support causality across events similar to what is provided by logical clocks while maintaining the logical clock value to be always close to the physical clock. The requirements of HLC can be captured formally as follows.

Let pt be the physical time at any node in the distributed system. We use $pt.e$ to denoted the physical timestamp at the node where the of event e happens. Given a distributed system, assign a timestamp ℓ for every e, denoted $\ell.e$, such that:

- For any two events e and f, $e \rightarrow f \Rightarrow \ell.e < \ell.f$.
- The timestamp $\ell.e$ can be stored in space of $O(1)$ integers
- The timestamp $\ell.e$ is represented in bounded space, and
- $\ell.e$ is close to $pt.e$, in symbols, $|\ell.e - pt.e|$ is bounded.

The above requirements can be understood in the following manner. The first requirement captures the notion of causality in distributed systems. The second requirement ensures that updates to $\ell.e$ is possible in $O(1)$ operations and hence reduces the overhead of maintaining the logical time. The third requirement indicates that the logical time has a bounded space requirement and does not depend on parameters that can grow unbounded such as the number of nodes in the system. The last requirement captures the constraint that $\ell.e$ is close to $pt.e$ which enables HLC to be used in place of physical time. This decoupling from physical time allows applications the replace using physical time with hybrid logical clock.

HLC maintains its logical clock at every node j as two variables $\ell.j$ and $c.j$. The tuple $\langle \ell.e, c.e \rangle$ is the logical timestamp assigned to event e. The first variable $\ell.j$ is used to maintain the maximum of pt information that j learnt so far. The second variable $c.j$ is used to capture causality updates when the ℓ values of two events are equal. This separation allows site j to *reset* $c.j$ when the information about maximum pt that j is aware of equals or goes beyond $\ell.j$. Notice from the description that for two events e and f at a node j, it is possible that $\ell.e = \ell.f$. Thus, unlike logical time, ℓ need not be incremented for every event. This feature ensures that one of the following holds: (1) a node receives a message with a larger ℓ resulting in an update to ℓ and a reset of c, or (2) if the node does not get any messages from other nodes, then its ℓ stays the same, and its pt will increase naturally which results in an update to ℓ and reset c.

The algorithm HLC uses is shown below. Initially, at all nodes j we set $\ell.j = 0$ and $c.j = 0$. When executing a local event e, including a send event at node j, node j updates ℓ and c as follows.

- Set $\ell'.j = \ell.j$
- Set $\ell.j = \max\{\ell.j, pt.j\}$
- If $\ell.j == \ell'.j$, then $c.j = c.j + 1$ Else $c.j = 0$.
- Set the timetstamp of e as $ts.e := \langle \ell.j, c.j \rangle$.

When executing a receive event of message m at node j, it does the following.

- Set $\ell'.j = \ell.j$
- Set $\ell.j = \max\{\ell.j, \ell.m, pt.j\}$

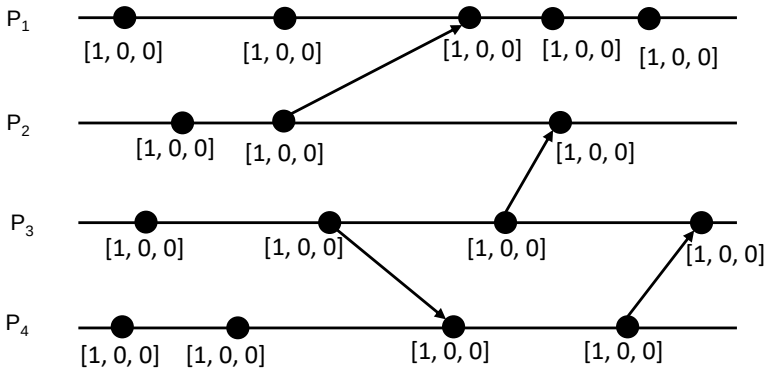


Figure 2.7: An illustration of Hybrid Logical Clocks.

- If $(\ell.j == \ell'.j == \ell.m)$, then $c.j = \max\{c.j, c.m\} + 1$ Else if $\ell.j == \ell'.j$, then $c.j = c.j + 1$ Else if $(\ell.j = \ell.m)$ then $c.j := c.m + 1$ Else $c.j = 0$
- Set the timetstamp of e as $ts.e := \langle \ell.j, c.j \rangle$.

From the above description, we notice that when a new send event f is created at node h, $\ell.j$ is set to $\max(\ell.e, pt.j)$, where e is the previous event at j. Such an update to ℓ ensures that $\ell.j \geq pt.j$ at all times. However, looking at events e and f at node j, it is possible that $\ell.e = \ell.f$. The increment to the second part of the timestamp tuple helps ensure that $\langle \ell.e, c.e \rangle < \langle \ell.f, c.f \rangle$ when compared lexicographically. If $\ell.e$ differs from $\ell.f$ then $c.j$ is reset, and this allows us to guarantee that c values remain bounded. Figure 2.7 shows an example of using hybrid logical clocks.

Similarly, to process a receive event at node j, $\ell.j$ is updated to to $\max\{\ell.e, \ell.m, pt.j\}$. Depending on whether $\ell.j$ equals both of $\ell.e, \ell.m$, or neither of them, $c.j$ is set accordingly.

The following theorems prove that the HLC construction satisfies the four conditions needed.

Theorem 3 *For any two events e and f, if $e \rightarrow f$ then $ts(e) < ts(f)$.*

Proof: Notice that comparing timestamps in HLC is done via the lexicographic ordering of the tuple $\langle \ell.e, c.e \rangle$. (In particular, we say that $\langle a, b \rangle < \langle c, d \rangle$ if and only if either $a < c$ or $a = c$ and $b < d$.)

Based on how the timestamps of events are assigned, the theorem holds easily. □

We move to see how HLC meets condition 4 via the following sequence of theorem.

Theorem 4 *For any event e, $\ell.e \geq pt.e$.*

Theorem 5 *Let e be any event. Then, $\ell.e$ denotes the maximum clock value that e is aware of. In particular, $\ell.e > pt.e \Rightarrow (\exists f : e \rightarrow f \wedge pt.f = \ell.e)$.*

Proof: The proof can be done by induction on events. □

Let ϵ denote the physical clock synchronization uncertainty. Notice that ϵ depends on physical factors including the protocol such as NTP used to synchronize physical time. We now use Theorem 5 and the notion of ϵ to show that $\ell.e - pt.e|$ is bounded for any event e.

Theorem 6 *For any event e, $|\ell.e - pt.e| \leq \epsilon$.*

Proof:

Due to the nature of ϵ, we cannot have two events e and f such that $e \rightarrow f$ and $pt.e > pt.f + \epsilon$. Now, appealing to Theorem 5, we get the desired result. □

We now show that HLC satisfies condition 3 using once again the causuality among events.

Theorem 7 *Let e be an event with $c.e$ be $k > 0$. Then, there exists a sequence of k events $f_1, f_2, \cdots, f_k$ such that (i) for $1 \leq i < k$, $f_i \rightarrow f_{i+1}$, and (ii) for $1 \leq i \leq k$, $\ell.f_i = \ell.e$, and (iii) $f_k \rightarrow e$.*

Proof: The proof of the theorem follows via induction. □

Theorem 7 immediately gives the following observation which limits the nature of events that lead to a high value for c.

Observation 8 *For any event e, $c.e \leq |\{f | f \rightarrow e \wedge \ell.f = \ell.e\}|$.*

Finally, we get the following bound on c.

Theorem 9 *Let N denote the number of nodes in the distributed system. For any event e, $c.e \leq N \cdot (\epsilon + 1)$.*

Proof: We appeal to Observation 8 and obtain that $c.e$ has a value bounded by the number of events f that precede e and have the same ℓ value. Note that f can occur at any node in the system. Further, from the construction of HLC, it holds that $\ell.f \geq pt.f$ for any event f. Finally, due to the

clock synchronization constraint and the fact that $e \rightarrow f$, we have that $\ell.f \leq pt.e + \epsilon$.

From the above, we note that the events f that satisfy $f \rightarrow e$ and $\ell.f = \ell.e$ are those that occur at some node when the physical time at that node is in the interval $[\ell.e, \ell.e + \epsilon]$. Since the physical clock is incremented by at least one for every event, there are at most $1 + \epsilon$ such events at any node. With N nodes in the system, the bound on $c.e \leq N \cdot (\epsilon + 1)$ follows. □

In summary, what HLC offers is a novel way to decouple physical time and logical time yet obtain guarantees on the window of uncertainty. Morever, HLC does not rely on any expensive or special purpose mechanisms to synchronize physical time, nor does it rely on any particular physcial time synchronization protocol. HLC uses the physical time but does not update the physical time at any node. The update to physical time is left to the physical time synchronization protocol. However, udpates to the logical timestamp follow information obtained through timestamps on messages received, which are a function of the physical time at the sender.

Finally, HLC is not the silver bullet to solve time the problem of time synchronization in distributed systems. HLC still includes a window of uncertainty and applications may need to wait out this window of uncertainty, just as in using TrueTime [29].

2.7 Leap Seconds and Smearing

Depsite all the careful theoretical and engineering efforts, there is still one issue that most practical systems face with. Notice that time is also a measure of how the Earth revolved around the sun. Changes in this speed occasionally or as a continuing pattern can cause changes in the accuracy of the time systems we use. It is observed that the speed with which the Earth revolves around the Sun varies slightly due to several factors. This results in a difference of roughly one second over a couple of years between physical clocks and the time as measured by the rotation of the Earth around the Sun. To adjust for this difference, the international body General Conference on Weights and Measurements (CGPM) [2] adopted the practice of adding or removing one second from the physical clocks. The last such additon to the local time at Hyderabad happened in June 2017. This extra second added or removed from physical clocks is called as *leap second*.

Systems such as TrueTime also adopt the model of adding a leap second as needed. One additional technique that is used is to spread the change

acorss multiple instances. This practice goes by the name of leap second smearing.

Questions

Question 2.1. Set up a small experiment to simulate NTP in a LAN environment. You can scale the simulation to systems running across larger physical distances by introducing a delay in proportion to the physical distance plus some noise.

Question 2.2. In logical scalar time and logical vector time, is the increment d needed to be common across pocesses in the distributed system? Justify your answer.

Question 2.3. In the context of the Hybird Logical Clocks, if the time for message transmission is long enough so that the physical clock of every node is incremented by at least d, where d is a given parameter, redo the bound on $c.f$ for any event f. From Page 6 of the HLC paper.

TO MOVE TO EARLIER PART:

Modern distributed systems is also about creating large-scale highly reliable, fault-free, and avaialble systems using inherently unreliable and limited in scale with additional challenges including degrees of asynchrony and uncertainty.

All problems in computer science can be solved by another level of indirection. –David Wheeler

no changes.

Chapter 3

Distributed Data Storage

Distributed systems typically comprise a large number of nodes. This means that embody a lot of storage space. This space can effectively be used to store a large amount of data. We need to understand that bespoke storage systems are expensive, difficult to maintain and are difficult to scale. It is a much easier to take regular machines and servers, connect them together and treat the ensemble as one large storage unit. Users or clients can be provided a single standardized interface to query this system. The storage model can be really simple. It needs to present itself as a simple key-value store, where the *key* is the name of the file and the *value* is the contents of the file.

There are many ways to create such large distributed systems. We can use a large array of cheap servers with inexpensive storage devices or we can co-opt the machines of regular users. An example of the latter type of system is BitTorrent where the client machines are used to store some data as well. Regardless of the architecture of the system, the fact is that storing a large amount of data on a distributed system is reasonably mature as of today and has proved to be extremely useful.

Such systems consist of a large number of machines that play identical roles. They are thus referred to as "peers" and such distributed systems are consequently known as peer-to-peer (P2P) systems. P2P systems need not be purely distributed systems, they can have some degree of centralization. Over the last three decades, several generations of P2P systems have been proposed. There are at least three major generations and many believe that each major generation can be subdivided into many minor generations.

Broadly speaking, the first generation of P2P networks used some degree of centralization. There were some servers that were tasked to store index

files. An *index file* maintains a mapping between file names and machines that store them. While looking up a file name, it is necessary to fetch the corresponding index file and then fetch the contents of the file. There are many problems with such centralized systems. The nodes that store the index files become a performance bottleneck, make it hard to scale the system and also become a legal liability (as was the case for Napster).

The first generation was quickly replaced by the second generation of P2P systems. These systems were true distributed systems. The peers worked collaboratively to locate keys and disseminate information. Such systems came into two different types. They could either be unstructured, where there are arbitrary connections between the nodes in the network. Information is disseminated using protocols that are akin to gossiping or epidemic propagation. There are more efficient methods of information dissemination where we assume some structure in the network. Regardless of the physical connection, it is possible to logically connect the nodes by creating a virtual network (network over a network). Such networks are known as *overlay networks*. Ring-based overlays are common. We shall show in Section **??** that such overlay-based networks can be very effectively used for information storage and dissemination.

Finally, we shall discuss 3^{rd} generation P2P networks, where the focus is on security. The receiver does not know where a file is stored and the sender does not know who the receiver is. Such networks protect the identities of all the parties involved in the file transfer process. We shall see that such networks had to be realized to circumvent legal issues.

3.1 1^{st} Generation P2P Networks: Napster

Let us start our discussion with first generation P2P networks. We will discuss the most popular network in this category, the Napster system.

3.1.1 Napster

Let us go back to 1999. At that point of time, the MP3 file format had become very popular. The reason was that it suddenly became possible to store a 5 minute song using 5 MBs of storage. The quality of the sound was very nice and clear. Your authors clearly remember that at that point of time, streaming videos or even storing videos in small pocket-size devices was unthinkable. However, storing audio had become possible and thus MP3 players were very popular. They could fit in pocket-sized devices called MP3 players that especially found a lot of use while exercising. They became one

of the most visible devices in gyms. Major record labels started distributing their songs in the MP3 format. Sites like mp3.com also sprung up. They enabled the seamless sharing of MP3 files.

At that point of time, Shawn Fanning realized that it would be great if there was a search engine that was dedicated to MP3 files only. He envisioned a larger distributed system that would allow users to exchange or trade MP3 files. Along with exchanging MP3 files, the engine could help users connect with each other and make friendships. This is how the idea of Napster came about. Think of it as a combination of a file sharing service and a social network. In terms of its implementation, Napster was a client application that was installed on a client machine that connected to a central server, known as the *broker*. Napster is now a well-recognized protocol even though it has been replaced by newer technologies that solve the problem in a much better fashion.

Basic Operation

All that a user has to do is open the Napster utility and log on to a broker. Client machines that host users are expected to share the list of MP3 files that they store with the broker. Brokers collate the set of files obtained from different clients and create an index or a directory.

The first action taken by the broker after a client logs in is to fetch the list of MP3 files that it has. After this, the client machine is free to search for an MP3 file on the broker. All that the user has to do is enter a search term (query term). If the file is present on some other client machine and it features in the broker's list, the broker responds with the IP address of the client machine that contains the MP3 song. The broker provides a straightforward index or directory service. The files per se are transferred between client machines after getting the IP address from the broker. The client needs to establish a connection with the serving client machine and get a copy of the song (MP3 file). This is shown in Figure 3.1, where we can see multiple client machines and a broker. Note that the file transfer happens between the client machines without involving the broker.

Figure 3.1: The Napster network

Napster Protocol

Let us now look at the Napster protocol. The broker runs on port numbers 7777, 8888 or 8875. The message format is quite simple. Any Napster message has three fields namely the length, type and data. The *length* field stores the length of the message in bytes. Two bytes are allocated for storing the length. There can be several types of Napster messages such as error, login, login acknowledgement, version and upgrade messages. Finally, the *data field* corresponds to the actual message. The data represents the contents of the MP3 file.

In Napster, along with broker nodes, we have client machines and lookup servers. The client first finds the address of a broker by contacting a lookup server. The addresses of lookup servers are globally known, whereas the addresses of brokers need to be provided by the lookup server. There is some opportunity for load balancing here. The lookup server can find the least-loaded broker. To use this mechanism, the client needs to exchange messages with the lookup server using the regular TCP/IP protocol. The broker subsequently is ready to process queries that are received from clients.

Five Types of Napster Processes

A Napster peer runs five kinds of processes in parallel. There is a main coordination process whose job is to connect and communicate with brokers. This is also known as the *main coordination instance*. The *listener* instance handles incoming connections from peers. The *upload*, *download* and *push* instances are used to transfer files between peer machines.

The state diagram of the main coordination instance is shown in Figure 3.2.

Figure 3.2: Actions of the coordination instance

Let us start from the offline state. The first action is to connect to a Napster lookup server. After the connection is set up, the state transitions to the find best broker state. T The next task is to find the least-loaded broker. In this regard, the lookup server is free to use any kind of load balancing algorithm. Once a broker is found, the login process is activated and the state transitions to online (after successfully logging in). In this state, the client machine can send queries to the broker and receive responses. It can also send metadata to the broker. The metadata contains the list of files

that the client has. Each row contains information such as the name of the file, the title of the song, names of the artists, etc.

The download instance starts from the download state. It then sends a download request to the broker and awaits a reply. There are many reasons for the reply to be denied at this stage. The request may be illegal, the remote host (remote peer) may not be able to upload the file, or the file may simply not be there. If the file can be downloaded, then the broker sends the download ack message. The contents of this message contain the following: location of the file (hostname, IP address) and the TCP port.

There is an interesting twist here. Let us name the requesting client as U and the client that has the file as V. In the normal sequence of events, U sends a request to V or in other words U initiates a connection with V using the TCP port that the broker has provided to it. V can either accept or deny the connection. If it accepts the connection, it scrutinizes the request and if all is well then it provides a copy of the requested file to U. All of this happens in the remote client download state.

However, the twist here is that V may be in a private network where all the network addresses are only locally visible. It is not possible to make a connection to V directly from outside the network. In this case, the TCP port that is returned by the broker is 0. As a result, U enters the remote client upload state. In this case, V needs to initiate the process of contacting U and the broker needs to facilitate this. V thus sends a connection request to U and once it is accepted, it sends the metadata and then the contents of the MP3 file. The sequence of actions is shown in Figure 3.3

Figure 3.3: Remote client upload instance

The last instance is the remote client download instance. It does not have the complexities of the earlier instance. It directly initiates a connection with a remote peer and downloads the file.

Legal Issues

Napster was for its time a revolutionary idea. It suddenly allowed users all over the world to freely access MP3 files. This was clearly illegal. Music is owned by its creator. It is clearly not meant to be freely distributed all over the world, that too without paying any money. It is true that Napster did start a revolution in internet-based content access, but it was clearly not on the right side of the law.

Moreover, its centralized architecture had problems. It was hard for clients to anonymize themselves. They had to make their identities visible to the server (central site). The central site had all the control, was aware of all the client's identities and was a single point of failure. Clearly, a scalable distributed system should not rely on a central site.

Regardless of the technical shortcomings, the legal issues stood out and ultimately led to its downfall. By the late nineties, it was neck deep in its legal troubles and was sued by multiple music bands including the Recording Industry Association of America. It finally closed its service in 2001 and filed for bankruptcy in 2002.

Now, looking back at Napster in 2024, we get mixed feelings. The fact that it allowed free sharing of potentially copyrighted music and created financial losses to music creators is mostly true. However, here also there is a school of thought that allowing free sharing of MP3 files of the internet actually boosted their sales. Nevertheless, centralized file sharing has its drawbacks and got phased out for techno-legal reasons.

This led to the second generation of P2P networks that did not have a central server. Instead, all the machines were peers. They followed elaborate protocols to exchange information between them and provide the abstraction of a large data store especially a hashtable like key-value store.

3.2 2^{nd} Generation P2P Networks

Second generation peer-to-peer networks are noted by the absence of a central server. The nodes share information with each other as peers. There are two important categories of algorithms in this space. The first class of algorithms work on unstructured networks. Here we assume that nodes can be arbitrarily connected to each other. Protocols rely on classic epidemic propagation models to disseminate information via gossip-based mechanisms. These approaches are effective in the general case.

However, their performance is limited. The performance can be improved by considering structured networks where the nodes are arranged as a mesh or ring. Even if the physical network does not resemble a mesh or ring, a virtual network called an overlay can be created to reflect structured connections. A structured overlay allows us to design bespoke protocols that can be used by nodes to quickly locate stored data items and update the distributed data store. There are several sub-generations of solutions in this space. There are protocols like Gnutella that arose out of gossip-based algorithms. This later was supplanted by fast algorithms that exclusively

rely on circular ring-shaped overlays. A lot of work has been done in this space, which are known as distributed hashtables. There are two important proposals named Pastry and Chord that are heavily used. BitTorrent is one of the most popular protocols in this space that combines many of the aforementioned elements. It has some weak anonymity features.

Third-generation peer-to-peer networks expand on the anonymity aspect. They provide very strong privacy and anonymity guarantees for both file servers and recipients.

3.2.1 Gossip and Epidemic Algorithms in Unstructured Networks

A classical problem in unstructured networks is to simply broadcast a value to the rest of the nodes. This is a simple problem if we consider structured networks such as a ring or a tree. For example, in the case of a ring, all that we need to do is send a message on one side of the ring and wait for it to move a full circle and come back to the starting point. In the case of a tree, all that a node needs to do is forward the message to its children. However, in an unstructured network where there is no such predefined structure, sending a message to the rest of the nodes is a difficult problem. We need to ensure that there is no exponential blowup in terms of the number of messages that are sent.

There are three generic approaches in this space.

Direct Mail In this case, the ids of all the nodes in the network are known. For example, if the ids are IP addresses, then all that needs to be done is send messages individually or multicast them (wherever possible) to the rest of the nodes. This can degenerate to a sequential process. The node sending all these messages can become overloaded and become a point of contention. Nevertheless, this approach known as *direct mail* has its share of benefits especially when the network is small and all the nodes in the network are known to the sender.

Anti-Entropy In this case, a node need not have perfect knowledge of the network. It only needs to know the ids of some of the other nodes. It can choose one of them random and mutually synchronize its contents with that node or just send its updates to that node. For example, if a message needs to be sent and that message is not there in the log of the other node, then the message can be transferred to the log of the other node and it can be instructed to propagate the message further

in the network. This is a very effective protocol and is still heavily used in many commercial settings as of 2024.

Rumor Mongering The problem with anti-entropy is that it is a protocol without brakes – it simply keeps going on without terminating. Another way of broadcasting an update is called *rumor mongering*, where a node sends updates to other nodes and when it sees that most of its neighborhood is already *infected* (possesses) with its update, it reduces its aggressiveness in sending the update and ultimately stops. This protocol ensures that the overall process can ultimately come to a stop and by that time we can guarantee that at least most of the nodes have received the update. This is an example of a complex epidemic, which is arguably faster than the anti-entropy protocol. Sadly, it does not guarantee that the update will reach all the nodes. It is indeed possible that we terminate the protocol before all the nodes have received a copy of the update.

Let us now model this process mathematically in a system with n nodes. The modeling proceeds in roughly the same manner as we model epidemic propagation. Propagating an update in an unstructured network is quite similar to an epidemic propagating in the human population. There are *infective* nodes whose job is to propagate the update, there are *susceptible* nodes that have not received the update up till now and then there are *removed nodes* that are not involved in the protocol at all. We assume that the protocol proceeds in rounds. In each round, all the infective nodes send updates to one or more nodes (typically 1).

Mathematical Modeling of Anti-entropy

There are three different kinds of information exchange in an anti-entropy protocol (refer to [35]). They are known as *push*, *pull* and *push-pull*, respectively. In the push mode, the update is simply sent to a target node, and if the target already has the update, then it discards the message. On the other hand, in the pull mode, a node proactively requests other nodes to send it any updates that they may possibly have. It simply adds all the new updates that it receives to its local log. The push-pull interaction does both. It *synchronizes* the contents of two nodes – it ensures that they have the same set of updates after the push-pool operation. Here we are assuming that every update has a timestamp associated with it and there is a notion of global time such that we can achieve a total order of all the updates. Even if we rely purely on local time, the same can still be achieved using Lamport

clocks. Let us thus make our life easy and assume that a notion of global time exists. This is however not an absolute necessity and the protocol can run perfectly well without such an assumption.

Let us now explain the basics of epidemic theory and provide a mathematical foundation for understanding anti-entropy protocols.

Consider a pull-based algorithm. Let p_i be the probability of a node remaining susceptible (not having the update) after i rounds. In the next $((i+1)^{th})$ round, it will not remain susceptible if it is contacted by another node that has the update. If we assume that any node can contact the current node in the next round, then we can easily deduce that the probability of a node continuing to remain susceptible (refer to Equation 3.1).

$$p_{i+1} = p_i^2 \tag{3.1}$$

A node remains susceptible if it is already susceptible in the i^{th} round and is contacted by a susceptible node in the $(i+1)^{th}$ round. The result is the product of the two probabilities: both are equal to p_i. Being contacted by a susceptible node means that either a message was sent without any updates or no message was sent (both the scenarios are equivalent here).

This is a rapidly declining function. We have $p_{i+k} = p_i^{2^k}$, which approaches 0 very quickly if p_i is a small value.

Let us now look at push-based methods. The expected number of infective nodes after i rounds of information exchange is $n(1-p_i)$. The probability of not contacting a node is $(1 - 1/n)$. Hence, the probability of not contacting any infective node in the $(i+1)^{th}$ round given that a node is still susceptible in the i^{th} round is given by the following equation.

$$p_{i+1} = p_i \left(1 - \frac{1}{n}\right)^{(} n(1-p_i)) \tag{3.2}$$

We use the fact that $(1-1/n)^n$ tends to e^{-1} as $n \to \infty$. Thus, for large n and small p_i (low probability of being susceptible), Equation 3.2 can be simplified. The result is shown in Equation 3.3.

$$p_{i+1} = \frac{p_i}{e} \tag{3.3}$$

Let us look at the structure of the two results – $p_{i+1} = p_i^2$ for pull-based and $p_{i+1} = \frac{p_i}{e}$ for push-based algorithms. It is clear that for low values of p_i – when there are very few susceptible nodes left – a pull-based strategy is more effective. In fact, we can easily show that when $p_i < e^{-1}$, a pull-based strategy reduces p_{i+1} more than a push-based activity, where the ultimate

aim is to make it as close to zero as possible. This also aligns with common sense. When we are nearing the end of the protocol (last few rounds), a push-based strategy is unlikely to yield a lot of dividends because we don't know, which nodes haven't received the update yet. Sadly, everything is left to random chance. However, such nodes shall always find it very easy to locate a node that has the update and fetch the update in a pull-based strategy. This is precisely why a pull-based strategy works towards the end of the protocol when there are few susceptible nodes left.

Let us now understand what happens at the beginning (when the anti-entropy protocol starts). When the information dissemination is starting, a pull-based strategy is unlikely to work because all the nodes that get contacted will most likely be susceptible themselves. On the other hand, a push-based strategy is guaranteed to disseminate the update far better. If we start with one infective node, that in the next round we will have 2, then 4 (with high probability), so on and so forth. The number of nodes with the update (infective) will continue to increase exponentially till a certain point. After that a situation will arise when most of the nodes that are contacted will already have the update. It is then a better idea to switch to a pull-based method.

It can be shown more formally that the growth in the number of infective nodes is multiplicative. As we have already seen, when the protocol is starting, the number of infective nodes keeps increasing exponentially. When the number of susceptible nodes reduces (low p_i), its rate of decrease approaches roughly $1/e$ if we continue to stick with a push-based method and if we opt for a pull-based method, then the rate of reduction is much sharper. In either case, we can conclude that given the multiplicative nature of this process, we expect it to complete in $O(log(n))$ time, where n is the number of nodes.

Rumor Mongering

Let us now consider rumor mongering, which is a complex epidemic. The information dissemination process gradually slows down and ultimately stops. In this case, we have three kinds of nodes: *infective*, *susceptible* and *removed*. The first two classes of nodes retain the same meaning as that in the anti-entropy protocol. The removed nodes no longer participate in the information dissemination protocol. For modeling such processes, typically differential equations are used (as we shall see next).

Let s be the fraction of nodes that are susceptible, i be the fraction of nodes that are infective and r be the fraction of nodes that are removed.

Clearly, $s + r + i = 1$.

The rate of decrease of susceptible nodes is proportional to the number of susceptible nodes itself and the number of infective nodes as well (refer to Equation 3.4). This follows from simplistic arguments because if we have a lot of susceptible nodes, then their count will decrease very quickly primarily because it will be easier for an infective node to contact them. On the other hand, if we have a lot of infective nodes, then they will be able to contact more susceptible nodes in a round. This will reflect in the rate of decrease of susceptible nodes. Kindly note that this is an approximate equation, because we are not taking into the consideration the fact that it is possible that different infective nodes may end up contacting the same susceptible node, or they may end up contacting another node that is infective. Nevertheless, approximate equations help us understand the broad structure of the solution. The trends become clear.

$$\frac{ds}{dt} = -si \tag{3.4}$$

These equations hold better when there are a lot of nodes and we are away from the extremes (all susceptible or most of them are infective).

The other aspect of rumor mongering is that nodes gradually lose interest in propagating the update. This is a good idea. Otherwise the algorithm will continue forever and like anti-entropy a lot of messages will get sent that actually reach infective nodes and thus do not serve a very useful purpose. However, the flip side of this is that the protocol may terminate too soon and we may miss infecting a lot of nodes. Equation 3.5 shows the loss of interest in propagating updates. Note the negative term $1/k(1 - s)i$.

$$\frac{di}{dt} = si - \frac{1}{k}(1 - s)i \tag{3.5}$$

The first term si indicates the rate of growth of the number of infective nodes. It is proportional to the number of already infected nodes and the number of susceptible nodes. The latter term – the number of susceptible nodes– is an approximation and basically serves to indicate that as the number of susceptible nodes increases, it is easier to infect more nodes in a round. We need to add another term that artificially damps this process – it can be delinked from the actual physical phenomena of update (epidemic) propagation as it is an artificial term, which is being extraneously added. Specifically, Equation 3.5 uses a damping factor k (higher it is, lower the degree of damping). In this case, the rate of damping is proportional to the number of infective nodes, which basically means that as the number

of infective nodes increases, the need to damp the process also increases proportionally. It is also proportional to the number of nodes that are not susceptible – they are either infective or they are removed. The aim is to explicitly factor in the number of nodes that are not susceptible anymore and use that as a separate variable for computing the degree of damping. It is important to note that the damping factor is something that we are adding to deliberately put slowdown the propagation process. It thus cannot be fully explained using the mathematics of epidemics.

We now have two equations here that need to be solved – Equations3.4 and 3.5. The solution is shown in Equation 3.6.

$$i(s) = \frac{k+1}{k}(1-s) + \frac{1}{k}log(s) \tag{3.6}$$

Here we are expressing the fraction of infective nodes i as a function of the fraction of susceptible nodes s. This is a complex function and there is a need to plot it to understand it. For different values of the damping factor k, the results are shown in Figure 3.4.

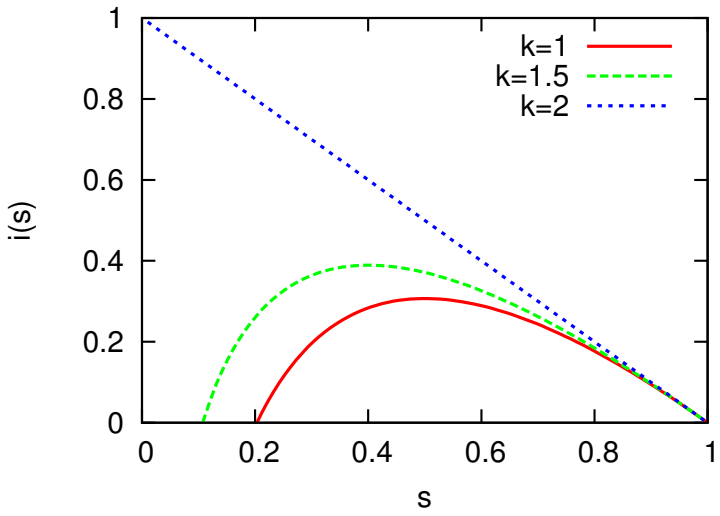


Figure 3.4: i as a function of s

Consider the case of $k = 2$ (lightly damped). We reach a point when no nodes are susceptible anymore – all the nodes are infective. We can assume that the plot starts from the point where no node is infective and all the nodes are susceptible (rightmost point on the x-axis). In the case of $k = 2$,

the number of infective nodes keeps on increasing roughly linearly until all the loads become infected.

This is however not the case with systems that are more damped, $k = 1.5$ or $k = 1$. In these cases, a situation indeed does arise when no node is infective yet there are some susceptible nodes still left. This means that because of the high degree of damping, the process terminates prematurely and because of that some nodes are left out – the update message does not reach them. Given that lower the value of k, higher the degree of damping, we find that the *residue* (susceptible nodes left out), is higher in the case of $k = 1$ as compared to when $k = 1.5$. This is on expected lines. We can see that as we move from $s = 1$ to $s = 0$, the number of infective nodes keeps increasing till a certain point. After that because of damping, this number starts to reduce and ultimately becomes zero. Due to the premature nature of the termination, the protocol leaves back a residue: lower the value of k, higher the residue.

It is thus important to understand that there is a trade-off between the number of messages that is sent and the residue that is left. We would ideally want the residue to be zero. However, we typically never have such an accurate understanding of the system. Hence there is always a little bit of under-damping or over-damping, which basically means that either we send many more messages than what is required or we leave a residue, respectively.

Let us now try to compute the residue. It is easily observed that when $i(s) = 0$, s is equal to the residue. It is given by the following equation.

$$s = e^{-\frac{k+1}{1-s}} \tag{3.7}$$

Equation 3.7 Is not a very convenient equation to solve. However, we can arrive at some general insights given the form of the equation. For example, we can observe that the residue is roughly an inverse exponential function of k. This basically means that as we increase k, we expect a substantial reduction in the residue.

Let us now study a few more fundamental results in this space. Let us try to find a relationship between the average traffic and the value of the residue. Let us say that there are m messages sent in the system per node, and there are a total of n nodes. Then we have a total of nm messages that are sent with the update. The probability that a given node misses all of these messages is $(1 - 1/n)^{nm}$, which as $n \to \infty$ becomes e^{-m}. In other words, the fraction of nodes that we expect to form the residue will also be the same fraction, i.e., e^{-m}. This can be proven with a simple probabilistic

argument starting from the basic notion of expectations.

Let us now understand this result. If m messages are sent (per node), and no message is received by a few nodes, then we expect the fraction of such nodes to be e^{-m}. For instance, if $m = 1$, then we expect roughly 37% of the nodes to miss the update. If $m = 2$, we expect 13.5% of the nodes to miss the update. For $m = 3$, this number becomes 5% (so on and so forth). m is clearly related to k – they are two different ways of quantifying the message throughput per node. There is clearly an inverse exponential relationship between the per-node traffic and the residue.

Given that rumor mongering can miss nodes, there is often a need to augment it with anti-entropy protocols such that at least all the nodes receive the update. It is often necessary to run the anti-entropy protocol towards the end of the rumor mongering phase such that any of the nodes that are left out can get the update in logarithmic time ($O(log(n))$).

Discarding Updates

3.2.2 Gnutella

Gnutella is a distributed network that does not rely on a central server. This is one of the key advantages of Gnutella over Napster. Because there is no central server, the onus of joining a network and even helping another node locate a file falls on peer nodes. This is more work but it is a more scalable solution and it does not have many of the legal issues that plagued Napster. A client node joins the network by contacting a known Gnutella host that is already there in the network.

The client can then send file search messages to its neighbors in the network. They in turn forward the message to their neighbors, so on and so forth. There is a possibility of an exponential blowup in terms of the number of sent messages. Hence, this process needs to slow down and ultimately be terminated (similar to rumor mongering). This is why a time-to-live field (TTL) is appended to every message. As the message is propagated through the network, this field gets decremented (once per hop) and ultimately when this field becomes zero, the node stops propagating the message. Before that point is reached, if a file server (a peer node in Gnutella) is able to locate the file in its file system, then it sends its IP address to the client. The client machine then directly establishes a connection with the node that has a copy of the file and starts a download process.

Akin to Napster, a Gnutella node runs many different processes (known as instances) that handle different aspects of the protocol. There is a con-

nection handler that manages connections with other Gnutella peers, a coordination instance that maintains the connections with other peer nodes, a download and upload instance, respectively. The latter two perform a similar task as their counterparts in the Napster protocol. The download instance handles the download from a remote node and the upload instance handles the upload to a requesting client node.

Requests in the Protocol

Initially, the connection handler is an offline state. Whenever there is a desire to connect to the network, it opens a coordination instance and sends a ⟨CONNECT⟩ message to another Gnutella peer. When the connection request is accepted, the state changes to online. Now at this stage, the client may try to contact other Gnutella nodes and also try to find the size of the network. This process of querying the Gnutella nodes and basically increasing the list of peer nodes is carried out in the ping state. Subsequently, when the client wishes to search for a file, it enters the search state. The ping state basically initializes the network state for a node that has recently joined the Gnutella network. Subsequently, the node can easily search for files by contacting live Gnutella nodes in its internal node list.

As we have already discussed, each message has a small TTL field. This ensures that the message is ultimately removed from the network and there is no flooding of messages. Messages also do not live in the network forever. With regards to transferring files, Gnutella faces the same problem as Napster, which is if the nodes are behind firewalls or are in a private network (NAT), then initiating contact with a node is not possible.

Let us consider three different cases: the server and client are both behind firewalls, the client is behind a firewall and the server is not, and the server is behind a firewall but the client is not. In the first case, it is not possible to establish a file transfer connection unless both of them are connected to an intermediary server at exactly the same time via their coordination instances. This is difficult and somewhat improbable, and thus the protocol does not consider this case.

If client is behind a firewall, then it is reasonably easy. The client can establish a connection with the server after it receives its IP address from the node that responded to the search reply. The client can then simply download the file. In this case it is like creating a connection with a regular website. Hence, there are no special cases here.

In the last case, if the server is behind a firewall but the client is not, then the client needs to request the server to establish a connection with

it. If the server is connected to some intermediary node via a coordination instance, then a message needs to be passed to it via that connection. The message should indicate that the server needs to initiate a connection with the client and transfer a given file. This is known as the ⟨`push`⟩ message. Given that the client is not behind a firewall, a connection can easily be established with it and the requested file can be uploaded.

The Coordination Instance

The coordination instance can receive five types of messages. We have already seen a few of them earlier. Let us now list down all of them.

The first is a ⟨`ping`⟩ message. After receiving this message, the receiver decrements the TTL field, and forwards message to other remote peer nodes. This message is primarily used for network discovery. When a node just joins a networks, it sends this message over the Gnutella network such that it can discover more of the network and find more peer nodes. This will be helpful when it is trying to search for files because it can send more search messages to its peer nodes.

The reply to a ⟨`ping`⟩ message is a ⟨`pong`⟩ message. This message is sent back to the node that just joined the network. This process basically works like a bidirectional handshake. The remote node also makes a note of the newly joined node. Given that all the nodes are peers of each other in the Gnutella network, it is also helpful for the remote node to be aware of the existence of the newly joined node.

Next, let us consider the ⟨`search`⟩ message. This message is sent with the details of the file (mostly audio files and videos). A remote node searches for the file locally in its database. Then it returns a copy of the file using the ⟨`search result`⟩ message. Otherwise, it forwards the message down the network after decrementing the TTL field.

As the name suggests, the ⟨`search result`⟩ message indicates that a given peer node has a copy of the file. It is important to note that at this point of time, only the existence of the copy of the file is noted and the same is sent to the requesting node. However, the node that has a copy of the file does not send the copy directly to the requesting node. This is primarily because the requesting node may have gotten positive search replies from several nodes. It cannot possibly accept copies of the file from all of them. This would not be very efficient, hence, the ⟨`search result`⟩ message is a very short message that simply has the identity of the remote peer node that can possibly supply a copy of the file. Subsequently, the requesting client node is expected to start a connection with one of the nodes that has

replied in the affirmative and download a copy of the requested file.

Note that in the latest official documentation [43] of Gnutella, the message names ⟨`search`⟩ and ⟨`search result`⟩, have been replaced with ⟨`Query`⟩ and ⟨`QueryHit`⟩, respectively.

Next, let us consider the ⟨`push`⟩ message, which is relevant only when the remote node that shall provide the file is behind a firewall. In this case, the requesting client sends the ⟨`push`⟩ message to the remote file-serving node via any open connections with other nodes that it may have. The remote node subsequently initiates a connection with the client node by sending a ⟨`giv`⟩ message. This allows the client to request for files. The request may contain the original file that it was searching for or possibly other files that the serving node is willing to share. Once, this list of files is shared, the remote node starts to upload the files to the client node using its upload instance.

At this point, the following doubt may arise. How do we send a message to a node that is behind a firewall and cannot accept incoming connections? The only way, as we have discussed, is to send a message to a node that has an open connection with the firewalled node. It can pass on the message via that open connection. A logical follow-up question is, "How do we find a node with an open connection with the firewalled node?" This can easily be ensured by following these rules while transmitting messages.

- A ⟨`pong`⟩ message follows the same path as the original ⟨`ping`⟩ message (reverse order of hops).
- A ⟨`search result`⟩ message follows the same path as the original ⟨`search`⟩ message (reverse order of hops).
- A ⟨`push`⟩ message follows the same path as the original ⟨`search result`⟩ message (reverse order of hops).

These rules ensure that if connections are kept alive for a slightly additional duration, then the reply message (⟨`pong`⟩,⟨`search result`⟩ and ⟨`push`⟩) can be sent back via the chain of open connections. By following this method, it is possible to reach nodes that are behind a firewall.

Comparison with Napster

Given that both the protocols, Napster and Gnutella, are reasonably simple and more or less achieve the same objective, we are in a good position to compare them. Napster has its share of problems in the sense that it is

centralized and it is possible to fix the legal liability. Napster nevertheless has some basic load balancing capabilities. There are ways of finding the least-loaded broker and ensure that no single server becomes a point of contention. Gnutella, on other hand, is more naturally load balanced. Even if there are network partitions, the protocol is still reasonably resilient. This is because as long as a node can at least contact some partition, it can get access to many files that are popular. So in a certain sense, the service still remains available. However, the quality may be *degraded* because the files in the inaccessible partitions will not be available for download.

In terms of traffic, both Napster and Gnutella do not do a very good job. Napster, of course, has more of centralization. Even if multiple brokers are used, the brokers can nevertheless become a point of contention. Furthermore, they also can be targets of denial-of-service (DoS) attacks. In a DoS attack, a malicious attacker can keep sending dummy requests and choke the network bandwidth or cause a server to crash. Gnutella is also more scalable and more immune to DoS attacks. However, a major concern with its operation stems from the fact that a large number of ⟨`ping`⟩ and ⟨`pong`⟩ messages are sent to discover the network. The same is true for the ⟨`search`⟩ messages that are sent throughout the network to search for a file. The reason for this is that Gnutella does not maintain a centralized directory. Hence, there is a need to send a lot of messages. The reason why there is no centralized directory is very clear – it is to avoid uncomfortable legal consequences. The price to pay is a large number of messages for network discovery and file search.

There is an issue related to the search quality as well in Gnutella. The search quality is better in Napster mainly because there are a fixed set of brokers that can be linked to each other, and they are guaranteed to have an entry for a file if it is there in the network. Of course, there are legal issues here, but at least we have one single central repository of files stored in one place. Gnutella does not have such a single directory. This avoid the obvious pitfalls of centralization – enhanced legal liability, single point of contention, reduce reliability and susceptibility to DoS attacks. It, instead, relies on a self-limiting form of message-broadcast to find nodes and files in the network. The TTL *limits* the number of messages. Nevertheless, the overall number of messages sent is much higher than Napster. There is another aspect to this other than the obvious problem of high message overheads – it can cause Gnutella to miss a lot of files. This reduces its accuracy, which more advanced protocols need to rectify.

3.3 Distributed Hash Tables

Creating an efficient distributed storage solution is a classical problem in distributed systems. In a large distributed system such as Amazon or Netflix there are millions of data items (search keys) and we also have thousands of requests per second for retrieving data: products for sale or movies (as may be the case). There is thus a need for a very high-throughput search mechanism that can handle *big data*: millions of keys and millions of values (each "value" can be a large object spanning several megabytes to gigabytes). There is thus a need to design a distributed data structure that can store such data across multiple nodes in a distributed system. There are many advantages of doing so. If we can distribute the requests across the nodes, then we can automatically handle the throughput and along with that if we can also distribute the objects across the nodes, then we truly have a *distributed solution* that can scale. Distributed systems also have natural advantages in terms of fault tolerance, resilience and all-round availability, which is an added benefit. Let us start with a discussion about classical hash tables and then extend the concept to distributed hash tables.

3.4 Classical Hash Tables

A classical hash table is a data structure that stores $\langle key, value \rangle$ pairs. Given a key, we can quickly find its corresponding value. The idea is to first convert the key to a uniformly varying random value and then find the index at which it is *most likely* stored in an array of values A. For instance, if we are storing the names of a list of students and their corresponding heights, we can use a hashtable. The key needs to get first *hashed* to an integer. Now, assume that the array has `S` entries. Let us introduce some terminology first. We hash the key k to compute a uniformly varying integer $\mathcal{H}(k)$. The index in the array where we search for the value will thus be given by $l = \mathcal{H}(k)\%S$. If multiple keys don't map to the same location, then we can check if the $A[l]$ is empty or not. If it is empty, then we can conclude that no value exists for the key in the hash table. Please refer to Figure 3.5.

Definition 10 *A hash table is a key-value store. Given a key, it quickly finds out if it exists or not. If the key exists, it returns the corresponding value. Hashtables typically take close to constant time as long as they are lightly loaded. To eliminate hash collisions the common methods are linear probing and chaining.*

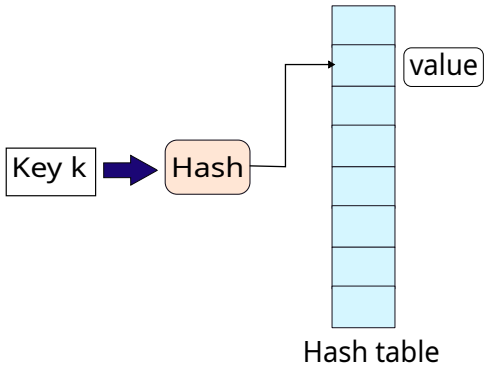


Figure 3.5: A generic hash table

However, if it is not empty, then we have a problem in our hands. The value that is present could correspond to key k or some other key k' that maps to the same array index l. This problem is known as *aliasing*, which can be mathematically represented as (% represents the modulo(remainder) operator):

$$\mathcal{H}(k)\%S = \mathcal{H}(k')\%S \tag{3.8}$$

If we for a second, assume that there is no aliasing, then a hash table turns out to be a very efficient data structure. We can fetch the value for a key or determine that the value does not exist in $\theta(1)$ time. It is thus ideal for a data structure that can be used to search for a key.

However, if we have aliasing, then there is a problem. We need to first ascertain if a value corresponds to a given key. Each cell of the array needs to store the value of the key as well. Now, if we have two keys that map to the same location in the array, then there are several options available to us: chaining and linear probing.

3.4.1 Chaining

Chaining is a simple solution where each entry of the array is a linked list of $\langle key, value \rangle$ pairs. All the keys map to the same array entry in this case. For searching for a given key, we need to iterate through the list. In the worst case, the search operation can take $O(n)$ time, where n is the total number of values in the hash table. This is because in the worst case, all the keys can map to the same array entry. This will not happen in practice because we use a uniform hashing algorithm that ensures that keys map to

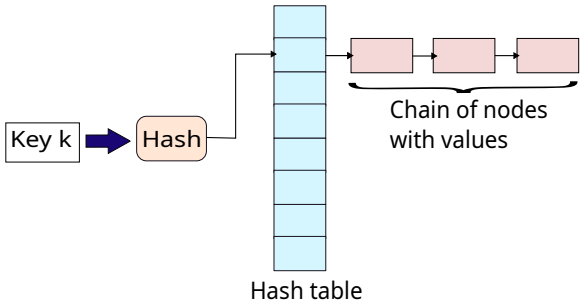


Figure 3.6: A hash table with chaining

all the entries roughly uniformly. We can also limit the size of the linked list if we are allowed to either discard entries or deny key addition requests to the hash table. If the maximum length of any linked list is α, then we will take at the most $O(\alpha + 1)$ steps.

3.4.2 Linear Probing

Other than complicating the data structure, we can adopt a simpler method. If the entry is not available in location l, then we search the array locations $l+1$, $l+2$, and so on. We can search the next κ entries $(l \ldots (l+\kappa-1)\%S)$. If an entry is found, then we can return the value, or we can return a null value (indicating that the key is not present in the hash table). Unlike the chaining algorithm, where we can store more keys than the number of array entries (`S`), we strictly store fewer keys than the number of array entries. Even though the worst case complexity is $O(\kappa + 1)$, we can deliberately create more aliasing. Hence, when the data structure is *sparse* (far fewer keys than the number of array entries), this method should be chosen.

3.4.3 Extension to Distributed Hash Tables (DHTs)

Consider a distributed system. We wish to use it as a hash table that stores a set of (key,value) pairs. The keys and values can be stored on different nodes. Insofar as an external user is concerned, it should not matter on which physical machine a key or a value is stored. The interface should look the same as a regular hash table. We would like the entire system to seamlessly scale with the number of keys and number of nodes. Any modern distributed system is extremely fault-tolerant and allows us to dynamically add/delete nodes at run time. This allows such systems to adjust to high

and low demand scenarios.

Definition 11 *A distributed hash table or a DHT is a distributed version of a hash table. It answers two queries:* get *(the value corresponding to a key) and* put *(insert a key,value pair).*

3.5 Consistent Hashing

We first review the ideas of consistent hashing that formalizes some of the aforementioned concerns and provides a framework to build distributed hash tables. The basic idea of consistent hashing [**?**] is to design a hash function that requires minimal changes as the range of the function changes. The solution proposed by consistent hashing has other advantages including (i) minimal need for messages to be exchanged by the caches, (ii) communication between caches can be asynchronous in nature, (iii) adapting seamlessly to the changing set of caches over time, and (iv) the ability of caches and users to work without access to full and current information about the set of caches.

As a step towards understanding the challenge, consider a single server that stores a large set of objects it intends to serve to users on demand. One possibility is to create a set of caches, distribute the objects across the set of caches via a hash function, and publish the hash function so that clients can use the hash function to locate the required objects. A good hash function can alleviate concerns such as distributing the objects uniformly across the set of caches. However, if the set of caches changes due to any reason such as failure, network errors, or otherwise. In this case, it is necessary to reassign objects currently assigned to such faulty caches. If this requires a change in the hash function as the range of the hash function, this may mean that a large portion of the objects have to be relocated.

Towards formalizing consistent hashing, we start with the following definitions. Let us imagine a distributed hash table with $\mathcal{I}$ as the set of items and $\mathcal{B}$ as the set of buckets. We let $I := |\mathcal{I}|$ and $B := |\mathcal{B}|$. We say that a *view* is a nonempty subset of $\mathcal{B}$. The view of a client corresponds to the set of caches that the client is aware of.

We call a function f as a ranged hash function if f has the domain $2^{\mathcal{B}\times\mathcal{I}}$ and a range of $\mathcal{B}$. In other words, a ranged hash function maps every item I with a given view to the set of buckets. The bucket that an item $i \in \mathcal{I}$ gets mapped to according to a view $\mathcal{V}$ is $f(\mathcal{V}, i)$. To simplify this notation, we write $f_{\mathcal{V}}(i)$ to denote $f(\mathcal{V}, i)$. As a diligent reader would have noticed, in

the above definition of a ranged hash function, we should insist that given a view $\mathcal{V}$, for every item $i \in \mathcal{I}$, $f_{\mathcal{V}}(i) \in \mathcal{V}$.

We call a ranged hash family $\mathcal{F}$ as a collection of ranged hash functions. Finally, a random ranged hash function is a function chosen uniformly at random from a given ranged hash family. We say that a ranged has family provides a consistent hashing scheme if it has the following four properties.

- **Balance**: We call a ranged hash family $\mathcal{F}$ balanced if given a view V, a set of items, and a random ranged hash function from $\mathcal{F}$, the fraction of items mapped to any given bucket in $\mathcal{V}$ is $O(1/|\mathcal{V}|)$.

- **Monotonicity:** While balance is a property expected of any hash function and not just a random ranged hash function, monotonicity is a property we seek of random ranged hash functions. In words, let us consider that we have views $\mathcal{V}_1$ and $\mathcal{V}_2$ with $\mathcal{V}_1 \subseteq \mathcal{V}_2$. Thus, $\mathcal{V}_2$ is obtained by adding more buckets to $\mathcal{V}_1$. The monotonicity property insists that the items that are moved due having a view with more buckets are essentially moving from a bucket in $\mathcal{V}_1$ to a bucket in $\mathcal{V}_2$, and not from a bucket in $\mathcal{V}_1$ to another bucket in $\mathcal{V}_2$. In symbols, this is written as follows.

  A random ranged hash function $f \in \mathcal{F}$ is monotone if over all views $\mathcal{V}_1$ and $\mathcal{V}_2$ with $\mathcal{V}_1 \subseteq \mathcal{V}_2$, and for every item $i \in \mathcal{I}$, it holds that $f_{\mathcal{V}_2}(i) \in \mathcal{V}_1$ implies that $f_{\mathcal{V}_2}(i) = f_{\mathcal{V}_1}(i)$.

- **Spread**: Spread refers to the number of different values that a random ranged hash function has for an item i under a set of views. To that end, let $\mathcal{V}_1, \mathcal{V}_2, \cdots, \mathcal{V}_k$ be k different views with each view having at least of size $1/t$ of the buckets that are part of all the k views.

  For a given item $i \in \mathcal{I}$ and a random ranged hash function $f \in \mathcal{F}$, we define the spread of i, denoted $\sigma(i)$ as $\left|\left\{f_{\mathcal{V}_j}(i)\right\}_{j=1}^{k}\right|$. The spread of i under f is essentially the number of distinct value of f for i under the k views. Using $\sigma(i)$, we can then define $\sigma(f)$ as $\max_{i \in \mathcal{I}} \sigma(i)$ as the spread of a random ranged hash function $f \in \mathcal{F}$. Finally, a ranged hash family $\mathcal{F}$, has spread σ if the spread of a random ranged hash function from $\mathcal{F}$ has a spread σ with high probability.

- **Load**: Intuitively, the load condition is about how many items are mapped to any given bucket by a hash function. When we move to consistent hashing, this property requires a little big of additional rigor to address the question of clients having multiple views.

Figure 3.7: An example illustrating the approach of consistent hashing. Orange colored dots represent the positions of buckets and blue color dots represent the positions of the items. The portions of $[0, 1]$ that the vertical lines partition into correspond to the segment of $[0, 1]$ such that items falling in the bucket that owns this segment.

> To that end, let $\mathcal{V}_1, \mathcal{V}_2, \cdots, \mathcal{V}_k$ be k different views. Let b be a bucket in $\mathcal{B}$ and f be a random ranged hash function from $\mathcal{F}$. The load of a b under f, denoted $\lambda(b)$ is the quantity $\left|\cup_{\mathcal{V}} f^{-1}(b)\right|$. In other words, the load of a bucket is the number of items that are mapped to b under *any* view. Using the definition of $\lambda(b)$, similar to how we extended the σ, we say that the load of a hash function f, denoted $\lambda(f)$ is $\max_{b\in\mathcal{B}} \lambda(b)$. Finally, the load of a ranged hash family $\mathcal{F}$ is λ if the load of any random ranged hash function from $\mathcal{F}$ is λ with high probability.

As can be noticed, apart from balance, all the other three properties are beyond what a traditional hash function provides. A good consistent hash function should have low spread and load. In the following, we first show an example construction of a ranged hash family and use that family to create a distributed hash table.

3.5.1 Construction

Consider two random hash functions $r_{\mathcal{B}} : \mathcal{B} \to [0, 1]$ and $r_{\mathcal{I}} : \mathcal{I} \to [0, 1]$ that map buckets and items to a real number in $[0, 1]$. Let $\mathcal{V}$ be a view. We define the function $f_{\mathcal{V}}(i) := \text{argmin}_{b\in\mathcal{V}} |r_{\mathcal{B}}(b) - r_{\mathcal{I}}(i)|$. In words, the item i is mapped to the bucket b in $\mathcal{V}$ whose hash value according to $f_{\mathcal{B}}()$ is closest to the hash value $f_{\mathcal{I}}(i)$. For every choice of $r_{\mathcal{B}}$ and $r_{\mathcal{I}}$ we get a ranged hash function. The collection of these hash functions constitutes a ranged hash family $\mathcal{F}$. Figure 3.7 shows an example.

We comment on the range of the function $f_{\mathcal{B}}$ and $f_{\mathcal{I}}$. While the range is listed to be a real number in $[0, 1]$, we need enough precision in the range so that the hash value of any two objects can be distinguished. Doing so would require the range of these functions to have enough bits.

Another additional detail that is needed for the construction is about

having each bucket replicated $k \log n$ times where n is an upper bound on the number of buckets. In an implementation, we can visualize that each physical bucket is mapped to $k \log n$ real numbers in $[0, 1]$. The items that map to all of the $k \log n$ instances of a bucket are stored at the physical bucket itself.

In the following, we show that the family defined above is indeed a ranged hash family.

Theorem 12 *The family $\mathcal{F}$ is a ranged hash family and has the following properties.*

1. *$\mathcal{F}$ is balanced. In particular, for any random ranged hash function $f \in \mathcal{F}$, given a bucket b in view $\mathcal{V}$ and an item $i \in \mathcal{I}$, $\Pr(f_{\mathcal{V}}(i) = b) \leq \frac{O(1)}{|\mathcal{V}|}$.*

2. *The family $\mathcal{F}$ is monotonic.*

3. *If the number of views is $\rho \cdot I$ for a constant ρ, then for any item $i \in \mathcal{I}$, $\sigma(i) \leq O(\frac{\log I}{\rho})$ with high probability.*

4. *If the number of views is $\rho \cdot I$ for a constant ρ, then for any bucket $b \in \mathcal{B}$, $\lambda(b) \leq O(\frac{\log I}{\rho})$ with high probability.*

Proof:

Statement 1 concerning balance of $\mathcal{F}$ comes from the following. Recall that $f_{\mathcal{B}}$ and $f_{\mathcal{I}}$ are random hash functions. If $k \log n$ points are mapped to $[0, 1]$ by such a function, consider arranging the hash value of the buckets according to $f_{\mathcal{B}}$ in sorted order. Then it holds that the distance between any two consecutive hash values in the sorted order is $O(1/n)$ with high probability. See Question **??** for a proof of the above statement. This statement implies that the hash values of buckets partition the $[0, 1]$ interval into segments of length $O(1/n)$. Using this observation, and the nature of $f_{\mathcal{I}}$ it now holds that given a view $\mathcal{V}$ and an item $i \in \mathcal{I}$, the probability that i gets mapped to a given bucket $bin\mathcal{V}$ is at most $O(1)/|\mathcal{V}$.

Of the above statements, monotonicity of $\mathcal{F}$ is easy to show. Consider two views $\mathcal{V}_1$ and $\mathcal{V}_2$ with $\mathcal{V}_1 \subseteq \mathcal{V}_2$. Let f be a random range hash function from $\mathcal{F}$. Suppose that under the definition of $f_{\mathcal{V}}()$, an item i moves from a bucket $b_1 \in \mathcal{V}_1$ to a bucket $b_2 \in \mathcal{V}_2$. This happens because now the hash value of i under view $\mathcal{V}_2$ is closer to the hash value of $b_2 \in \mathcal{V}_2$. It follows that $b_2 \notin \mathcal{V}_1$. We conclude that therefore an item that is mapped to buckets in $\mathcal{V}_1$ would not be now mapped to another bucket in $\mathcal{V}_1$ according to the view $\mathcal{V}_2$.

Let us consider statement 3 concerning the spread of $\mathcal{F}$. Given a set of views, the spread of an item i according to random ranged hash function from $\mathcal{F}$ depends on the buckets that get mapped to $[0, 1]$ and are close to the mapping of item i. Since there is an upper bound of $O(\frac{\log I}{\rho})$ such buckets, with high probability, the spread of i is also upper bounded by the same number. Continuing with the definition of σ from $\sigma(i)$, we get the claim on the spread, additionally using Chernoff bounds [**?**].

Statement 4 on the load of $\mathcal{F}$ also follows from similar arguments. See Question **??** for an approach to prove Statement 4. □

The above discussion allows one to build a distributed hash table as follows. Pick two hash function $f_{\mathcal{B}}$ and $f_{\mathcal{I}}$ with their range in $[0, 1]$. The server will use the function $f_{\mathcal{B}}$ to map buckets to a point in $[0, 1]$ and uses the function $f_{\mathcal{I}}$ to map items to a point in $[0, 1]$ and uses these mappings to arrive at the cache that stores a given item. A client with a view $\mathcal{V}$ wanting to access an object i will compute $f_{\mathcal{V}}(i)$ to find the cache that should contain the item i according to its view $\mathcal{V}$. However, as the view $\mathcal{V}$ and the current set of caches may differ, the client may not find the object i in the cache computed according to view $\mathcal{V}$.

However, due to the monotonicity property of the construction, the item i is at the cache that is closest to the hash value of i according to the current view of the servers about set of caches. So, the client needs to find the cache that currently has i. This can be done by having auxiliary data structures. The paper by Karger et al. [**?**] builds a tree structure for this purpose where each segment of $[0, 1]$ that is not populated by a cache is a node in the tree. Other designs of distributed hash tables such as Chord [112] that we study later in this chapter use other mechanisms to help this process.

3.6 Distributed Hash Table: Pastry

3.6.1 Overview of Pastry

Let us now discuss one of the simplest DHTs, Pastry [103]. We start out by computing the hash of every node's IP address (or some other unique identifier). Any *uniform hashing function* ensures that the distribution of hashes is uniform. We create a 128-bit hash out of each node's unique information. We can safely assume that we will not have any aliasing. The probability of aliasing in any case is very low: 2^{-128}.

There are two ways in which we will interpret this 128-bit number. First, we will represent it in base 16. This means that any 128-bit number can be represented as a sequence of thirty two 4-bit (hexadecimal) numbers. This

specific aspect will be dealt with later. The more immediate aspect of this numbering is that if we have a large number (order of thousands), we can arrange all the 128-bit numbers (hashes of node ids) in a circular ring in an ascending order of numbers. This circular arrangement of nodes can be thought of as a virtual network or an *overlay*. An overlay is a network over a network where the nodes are logically connected to each other such that we can achieve a specific purpose. In this case, we are superimposing a circular overlay (ring) such that we can use it to implement a DHT.

The way that the circular overlay enters the picture is as follows. We also compute a hash of the key that we intend to search or add. Given the hash of the key and circular overlay of hashes, we need to map the key to a node on the ring. The key is mapped to the node whose hash value (read *node id*) is *closest* to the hash of the key. Henceforth, let us not prefix the term "hashes" before node ids and keys, and slightly abuse the notation in favor of simplicity. We shall henceforth use the term $nodeId$ or *node id* to refer to the hash of a node's unique information/location (e.g. IP address) and use the term *key* to refer to the hash of the key.

Definition 13 *An overlay is a virtual network. Every node is logically connected to a few of the other nodes.*

The overview of the *key search* process is as follows. We start with a node that is close to the node that is serving the request. A client machine is supposed to contact a server in the DHT, which initiates the process to search for the key. The server that receives the request starts out by comparing the length of the shared prefix between the key and its $nodeId$. It forwards the request to another server that has one more matching digit in the shared prefix. This way, the request gets closer and closer to the node whose $nodeId$ is closest to the key. If it is a search request, then the node searches for the key, otherwise if it is an *add* or *delete* operation then the node executes the corresponding operation.

In the aforementioned discussion, three important points have come up and thus we need three data structures to capture these points. The first is a *routing table* per node that contains a list of other nodes that have similar $nodeId$s and their details. The second is a neighborhood set (of nodes) and a leaf set. Both are a list of nodes. The *neighborhood set* captures the physical proximity of nodes with respect to the current node. This data structure is required to create/initialize the routing table. The third data structure is the *leaf set* that contains a list of nodes that have the closest $nodeId$s. Towards the end of the search process, it is important to iterate through the leaf set because the routing table ceases to be useful.

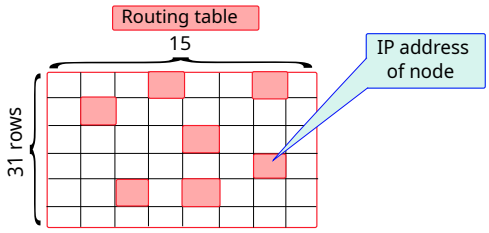


Figure 3.8: Pastry's routing table

3.6.2 Routing in Pastry

Figure 3.8 shows the structure of the routing table in Pastry. We have 32 rows and 15 columns. As indicated earlier, we represent a 128-bit id as a sequence of hexadecimal digits – each digit is 4 bits (based 16 hex number). Given that we have 128 bits in an id, we can represent it using 32 hexadecimal digits. Hence, we have 32 rows. Now, given a hexadecimal digit, we can have 15 different hexadecimal digits which are different. Now our routing algorithm basically tries to match the id of the key with the id of the node on the Pastry ring digit by digit. Hence, we need to have 15 columns, where each column corresponds to a different hexadecimal digit.

This is a very straightforward design. As we can see all that we need to do is to store a 32×15 matrix. The process of routing is also very simple. We start with the first digit of the key and the first digit of the id of the current node. If they match, then well and good, we proceed to the second digit. Assume that they don't match. Then we access the first row of the routing table and navigate to the column that corresponds to the first hexadecimal digit of the key. Let this digit be d. We would thus need to access the column corresponding to digit d. We expect to see a node's location stored there whose first digit is d.

Next, the protocol moves to a new node where we are sure that at least the first digit has matched. Now, we look at the second digit. The protocol in this case is also similar. If the second digit is equal to d', then we access the second row and go to the column that corresponds to the digit d'. If we find a node there (in the corresponding table cell), then we visit that node, and access the third row and so on. Gradually, we will match more and more digits, and the key will get closer and closer to the current node's id. Hence, as we observe, ultimately we will get very close to the node that is actually responsible for storing the key's value. Kindly note that in Pastry, our goal is to search for the node whose id is closest to the key. By this process, we

are essentially increasing the length of the shared prefix between the key and the node id by one in every routing step.

Up till now, we have been assuming the best case, which is that we always find a node at the cell that we are interested in. However, it is possible that the routing table may not be aware of any node that has a certain digit at that specific position. In this case, we need to do something else because clearly increasing the length of the shared prefix is not one of the options that we have. The other problem is that there is no clear-cut termination clause defined. We don't know, if we need to keep searching or we need to ultimately declare a node as the final result, which needs to contain the key.

This is where we need two additional data structures namely the *leaf set* and the *neighborhood set*. The leaf set $\mathcal{L}$ contains L nodes. Out of these L nodes, $L/2$ nodes have a smaller node id as compared to the current node's id. They are the closest node ids arranged in an ascending order. Similarly, the other half of the leaf set contains the closest $L/2$ nodes with larger ids (sorted in an ascending order again). The protocol is so designed that the research is accurate all the time. This basically means that at all points of time, we are aware of the closest L nodes that are the closest in terms of their ids.

The neighborhood set $\mathcal{M}$ contains M nodes that are the closest in terms of geographical proximity. We will see that this is also important particularly during a lookup operation because we don't want messages to travel very far. Geographical proximity ensures that message latencies are within limits. An important point needs to be made here. In any distributed system, we are not really concerned with the number of operations that happen within a node – they are deemed to be very fast. Instead, we look at the message complexity: the the messages that are sent to remote nodes. Sending a message is often a very slow operation and determines the critical path. This is why, we have a strong interest in ensuring that all the nodes that we visit are relatively close by.

Given that we have defined the leaf set and the neighborhood set, we are in a position to handle the two special cases that we discussed. We shall refer to Algorithm 1 in the subsequent discussion.

Progress Guarantees

The first question is what happens when we don't find an entry in the routing table. In this case, we are not in a position to increase the length of the shared prefix by one. However, we can still get closer to the destination node (the node that should contain the key). This is because we have the

leaf set. Assume that the key is greater than the node id. The leaf set is guaranteed to have $L/2$ nodes that have greater ids than the current node id. This means that we can always find a node within the leaf set, whose id is closer to the key than the current node. Let us consider the node that has the closest such distance between its id and the key and forward the current request to that node. Then, in this case, even though we are not increasing the length of the shared prefix, we are still reducing the distance (between the key and the node id). There is *forward progress*.

Again at that node, we will resume the process of routing. If we can increase the length of the shared prefix, well and good, otherwise we can continue to decrease the distance with the key. Having a full (or near full) routing table is always the better solution because we can rapid progress. However, in case the routing table has blank cells, we can still make progress as long as the leaf set has been initialized correctly. Note that as nodes enter and leave the system, the leaf set has to be updated continuously. Hence, the protocol will never get stuck and we will always be able to reach the destination node.

Ensuring Proximity in Accesses

This is where the neighborhood set comes handy. Recall that it is a set of M nodes that are closest to the current node in terms of geographical proximity. We can for instance quantify proximity as the latency of sending a message (ping times for example). When a node needs to join the network it contacts another node T that is close to it. Node T helps construct the routing table for the original node. It populates its routing table with all the nodes in its neighborhood set to start with. It then asks its neighbors for suggestions in for blank cells in the routing table. Given that the new node is close to T, it is also close to all the neighbors of T. Given that the routing table of node T was constructed in the same manner, it is also expected to have pointers to nodes in its routing table that are *close to it*. Hence, nodes from the routing table of T can also be used to populate entries in the routing table of the new node. This is how a degree of *locality* (proximity) is maintained.

A Look at the Formal Algorithm

Algorithm 1 shows the pseudocode for routing. The idea is to find the node that contains the value for key `K` (let this represent the hash of the key).

We first check if the key is within the range of the leaf set. If it is within

the range, then we forward the key to the node whose id is the closest to the key (Line 3). This is the termination phase of the algorithm. This means that we have found the node that needs to contain the key. Note that this operation relies on the accuracy of the leaf set. Often in a distributed system, we have many concurrent actions. As a result, it is often not possible to make very precise guarantees because of race conditions (concurrent events that many negatively influence each other).

Let us now come to the *else* part of the algorithm. We try to increase the length of the shared prefix by one. If we find a corresponding entry in the routing table, then we route the request to that node (as specified in the routing table's cell). Refer to Line 7. Note that the term $\texttt{K}_{\texttt{l}+\texttt{1}}$ refers to the $(\texttt{l}+\texttt{1})^{\texttt{th}}$ digit in the hash of the key.

Assume that this is not the case (the corresponding entry in the routing table is empty). Then we still need to ensure forward progress. We create a combined set comprising all the nodes in the leaf set and neighborhood set. We first try to see if we can increase the length of the prefix match (Line 10). If we can find a few such nodes, then we minimize the distance between the key and the node's id.

Otherwise, we find the node whose id is the closest to the key (see Line 12). Recall from our earlier discussion on forward progress guarantees that it is always possible to find such a node because of the way the leaf set is constructed. This means that we can always move to a new node where the distance between its id and K is lower than the corresponding distance for the current node. This guarantees forward progress, albeit slowly.

Remark 14 *In Pastry, forward progress is always possible. In every step, we get closer to the destination node.*

Algorithm 1 Pastry's routing algorithm

```
1: procedure ROUTE(Hash of the key (K), Routing table R, Leaf set L, Neighborhood set M )
2:     if L_{-L/2} ≤ K ≤ L_{L/2} then        ▷ K is within the range of the leaf set
3:         Forward K to L_i such that | L_i − K | is the least
4:     else
5:         l ← common_prefix(K, nodeId)
6:         if R(l + 1, K_{l+1}) ≠ null then
7:             Forward to R(l + 1, K_{l+1})
8:         else
9:             if There is a node u ∈ L ∪ M) such that prefix (u, K) ≥ l then
10:                Forward to a node v such that | v − K | is minimized
11:            else
12:                Forward to a node w ∈ L ∪ M ∪ R such that | w − K | is minimized
13:            end if
14:        end if
15:    end if
16: end procedure
```

A Note about Complexity

So, what is the complexity of the overall routing process? One may want to trivially argue that it is $O(1)$ because we can at most look at 32 digits (assuming we always find an entry in the routing table). However, it turns out that things are not that simple because we have a multitude of cases. Also, it is not necessary to use the routing table at every step, particularly when we get very close to the node that needs to contain the key. We basically need some more maths.

Let us start with a question. What is the probability that a hash does not have its first m digits in common with any other node's hash? Let us represent this probability as $P(m, n)$, where n is the number of nodes in the system. Let us write a quick list of points that will take us towards the solution. We will put a restriction on m here. We will assume that it is slightly more than $log_{16}(n)$ (details follow).

1. The probability that two hashes have their first m digits in common is 16^{-m}.

2. The probability that there is at least one mismatch is $(1 - 16^{-m})$.

3. The probability that the key does not have a prefix match of length m with all the nodes is $(1 - 16^{-m})^n$. Assume there are n nodes.

4. Let $m = c + log_{16}(n)$.

5. We have:

$$\begin{aligned} P(m,n) = (1-16^{-m})^n &= \left(1 - 16^{-c-log_{16}(m)}\right)^n \\ &= (1 - 16^{-c}/n)^n \qquad (\lambda = 16^{-c}) \\ &= \left((1-\lambda/n)^{n/\lambda}\right)^{\lambda} \\ &= e^{-\lambda} \qquad (n \to \infty) \end{aligned} \tag{3.9}$$

Let us take a deeper look at the expression $\boxed{\text{m = c + log}^{\cdot}16(\text{n})}$. c is the excess amount over and above $log_{16}(n)$. It is a constant that is added to the log term. Figure 3.9 shows the relationship between $P(m,n)$ and the constant c.

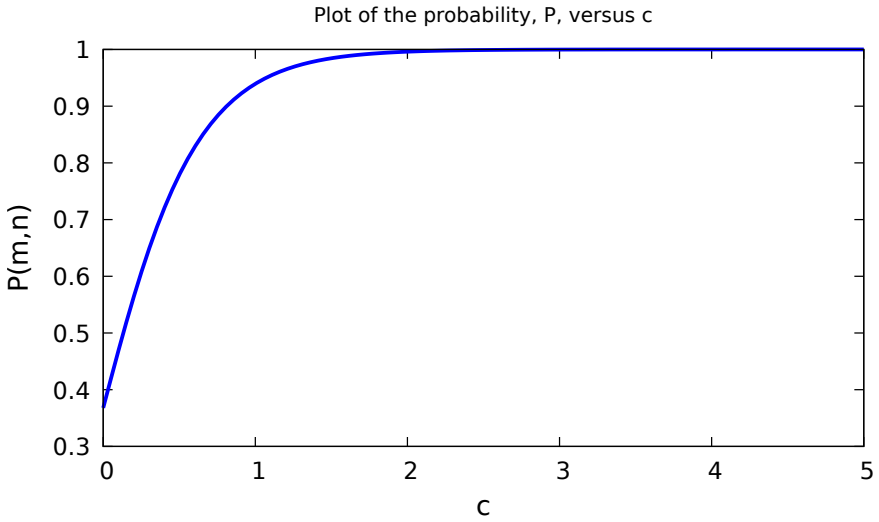


Figure 3.9: $P(m,n)$ versus c

We observe that $P(m,n)$ quickly saturates to 1 as c exceeds 3. This basically means that the maximum size of the prefix match that we can expect in a real setting is roughly $log_{16}(n)$. Beyond this number, the probability of not finding a match becomes near zero ($\because P(m,n) \to 1$). This

further means that the number of messages that we need to send because of matching entries in the routing table is limited to $log(n)$. Note that we have eliminated the base 16 here, because it is not relevant to our discussion any more.

Does this mean that the complexity (read message complexity) of Pastry is $O(log(n))$? Well, not yet. We have not considered the corner cases. A skeptic can always argue that the routing table will mostly be found to be empty, and as a result, the request will be sent to either a leaf node or a neighbor node (members of the leaf set and the neighborhood set, respectively). Let us now argue that this will not happen frequently enough to change the overall complexity from $O(log(n))$ to something else, subject to the fact that the routing table is well constructed (details follow).

We need to make some assumptions about the quality of the routing table. We need a broadcast mechanism to propagate node arrival information to all the interested nodes. We can define an *interested node* that has a long enough prefix match with it, let's say $log_{16}(n) + 3$. If we can ensure that using a combination of mechanisms, the routing table information is up to date and it is never the case that we find an entry to be empty, when it actually should contain a pointer to another node, then we can guarantee that we indeed complete the routing using $O(log(n))$ steps. It is possible for theoreticians to look at the degree of deviation from this ideal behavior that is tolerable, while still ensuring an asymptotic complexity of $O(log(n))$. Let us however not proceed in that direction. Let us assume that using a combination of protocols, the routing tables contain up to date information. Then we can safely say that routing in Pastry has a logarithmic complexity.

3.6.3 Node Arrival

The advantage of DHTs is that they can handle dynamic node arrival and departure. This allows them to automatically scale with the demand. For instance, if we have deployed a DHT in an e-commerce site, at the time of festivals, we can add more nodes. In the lean season, we can take nodes out of the DHT, and use them for some other purpose. In fact, this is why many companies that own large server farms start a cloud business on the side such that they can make good use of their servers when they are not required.

Node addition in Pastry is quite similar to the key lookup process. The node `X` that needs to be added computes the hash $\mathcal{H}$ of its node id and contacts a node `S` that is already in the DHT. Assume that there is a directory of nodes in the DHT that can be looked up. We can always choose

a node S that is geographically the closest to X based on the information that is present in this directory. Another method of achieving the same is to use expanding ring multicast, where we multicast a message with increasing time-to-live values. If any node is there in the DHT, it replies in the affirmative. This way we can quickly find the node S that is the closest in a geographical sense to X. The ping time can be a proxy for geographical distance here.

Initializing the State of Node X

Let the node that $\mathcal{H}(\texttt{X})$ is closest to be E. We ask X to start the lookup process. It then runs the regular routing protocol and finds a sequence of nodes from S to E. We now have a lot of information with us. We have access to leaf sets, neighborhood sets and routing tables of all of these nodes. All of this information can be used to initialize the state of node X. The neighborhood set of X can be initialized to the neighborhood set of S. Given that S and X are close by, this is the right thing to do. Similarly, the leaf set of E (the final node) can be used to initialize the leaf set of X. For the routing table, we can scan through all the entries in the leaf sets, neighborhood sets and routing tables of all the nodes from S to E and add them to the routing table of X.

Informing Other Nodes about X

Initializing, the state of X is not enough. We need to let the other nodes know about the existence of X. This means X needs to make an entry into the neighborhood set of S, and possibly other nodes in the neighborhood set of S based on proximity. It definitely should be there in the leaf set of E, and possibly in the leaf sets of some of the nodes that are in the leaf set of E. This means that the information about X needs to be broadcasted by E to all the nodes in its leaf set such that they can modify their leaf sets and routing tables accordingly. Finally, all the nodes in the path from S to E also need to modify their routing tables to include X.

Redistribution of Keys

The last step is redistribution of keys. By definition, a key is mapped to a node that its hash is the closest to. Now, assume that X is placed between nodes E and E′ on the ring. With X coming in, it is possible that some of the keys in E and E′ need to be moved to X along with their values. This redistribution of $\langle key, value \rangle$ data needs to be done.

Finally, after updating all this state, and redistributing keys, we will consider our work to be done. If you look at it, even though the overall complexity is $O(log(n))$, we still need to send a fair amount of messages and move a lot of data around. We would like to simplify this process to a large extent, if possible.

3.6.4 Node Deletion

There are two reasons why we would want a node to be deleted from the ring. The first is that it just voluntarily decides to leave on its own. The second is that it actually fails and other nodes in its leaf set or neighborhood set discover this fact. In both the cases, the node needs to be removed.

The Leaf Set

A node is supposed to periodically ping all the nodes in its leaf set and find if they are alive or not. If a node is found to have failed. Then, there is a need to repair the leaf set. This is not very hard to do. Let us say that a node with id greater than the current node's id is found to have failed. Then, we can just contact any other node whose id is more than the current node's id and fetch its leaf set. We can then use the information present in its leaf set to find additional nodes that can possibly be added to the current node's leaf set (based on the proximity of hashes). Let us say the current node's id is i. Assume that the node that failed had id $i + j$. We can then choose another node in the same direction (clockwise or anticlockwise) that has id $i + k$ and read its leaf set. We can then get the id of a node in its leaf set that we need to substitute the failed node with.

The Neighborhood Set

On similar lines, we can assume that nodes are periodically pinging nodes in their neighborhood sets. If a failure is suspected, then we can do something similar as the previous case (nodes in the leaf set). The current node can ask other nodes in its neighborhood set to send their neighborhood sets to it. It can then find a node that should be added based on proximity information. We should add the closest node that is not there in the set.

The Routing Tables

Given that a node has failed, a lot of routing tables will need to be repaired. The routing process also needs to be modified to take failed nodes into

account.

In general, it is a good idea to have multiple entries in each cell in the routing table. This provides a degree of robustness. If a node fails, we can always lookup other nodes. Moreover, if there is network congestion, we can then route the lookup message to an alternative node that can be accessed quickly. However, we may not be able to create so much of redundancy for all the cells in a routing table all the time. We need to consider the possibility of having a single entry in each cell, which we were assuming to be the case up till now. In this case, a node will contact the failed node, and it will not get a response. This case is no different from seeing an empty cell in the routing table. Recall that we have an elaborate protocol for handling such cases.

The other issue is to remove the failed nodes from routing tables. A node `X` that detects node `Y` to have failed can detect `Y` from its routing table. It can additionally try to do a lookup for `X` and start a string of messages. Each message needs to be annotated with the information that node `Y` needs to be removed. This way, we can cleanup some of the routing tables at least in nodes that are close to the failed node. Additionally, nodes that are close to `Y` (the failed node) can also check their leaf sets and remove `Y`.

Nodes can also proactively contact nodes in their routing tables and check if they are alive or not. If they are not alive, then a routing table cleanup procedure (as described in this section) needs to be commenced.

3.7 Distributed Hash Table: Chord

3.7.1 Overview

Let us now discuss another simple DHT algorithm that has had a lot of impact. As the name suggests, Chord [112] is related to the geometrical definition of a chord, which is defined as a line segment that connects two points on the circumference of a circle. Akin to Pastry, it also uses a hashing algorithm to map nodes to different positions on a ring-shaped overlay. It has inherent advantages in terms of simplicity and robustness as compared to Pastry. Consider an erroneous entry in Pastry's routing table. It could have been added maliciously or because of some bug in the system or the protocol's implementation. Then the message will be sent to a node that perhaps has a very low "prefix match". It will then take a long time for the message to come to the correct node. In the case of Chord, the protocol will recover very quickly. This makes Chord a very suitable choice for networks where we expect such kinds of bugs and system errors.

The basic idea is the same as Pastry – we have a circular overlay, where nodes are placed in an increasing order of hash values. There are however some differences. We don't map a hash to the node whose hash value is the closest. We instead map a node to the next node that has a higher id. Basically, given a hash we locate it on the circular overlay and then traverse clockwise (increasing ids) and find the next node that is there on the system. This is the node that is mapped to the hash, for instance the hash of the key.

Let us now look at the same activities such as routing, node addition and deletion (as we had explained for Pastry). In this case, we look at hashes bitwise as opposed to considering them as a sequence of hex digits. The bitwise treatment is not that inefficient as shall quickly see. In fact, in our view, it leads to a more elegant implementation. Let us assume that there are m bits in a node id (its hash basically). Let every node maintain a dedicated routing table, which we shall refer to as the *finger table*. The notion of *fingers* is very important in Chord. We shall formally define this concept over the next few paragraphs.

For each node, let us define two terms: *successor* and *predecessor*. For a given node on the circular overlay (just referred to as *circle* henceforth), let us define the next node (higher id) as the successor and the previous node (lower id) as the predecessor. The definition of the i^{th} finger is as follows for a given node with id n.

$$\begin{aligned} &finger[i].start = (n + 2^{i-1})\ mod\ 2^m, \quad (1 \leq i \leq m) \\ &finger[i].end = (n + 2^i - 1)\ mod\ 2^m \\ &finger[i].node = successor(finger[i].start) \end{aligned} \tag{3.10}$$

Figure 3.10 shows the set of starting positions of all the chords from a given node (with hash n). Then, Figures 3.11 and 3.12 show a few examples. They show nodes on an id circle and the fingers that they are mapped to.Figure 3.12 shows the mapping of a key to a succeeding node on the id circle.

As we can see, we have chords whose span increases successively by a factor of 2. They cover bigger and bigger parts of the id space. We shall see that this design where the sizes of the fingers increasing exponentially will help us design a very efficient search algorithm. A *finger* is thus a partition of the id space. Let us list the span of fingers in a tabular form (refer to Table 3.1). In the table we perform all the additions modulo 2^m. For the sake of readability, we omit the term $mod\ 2^m$ in each expression; however,

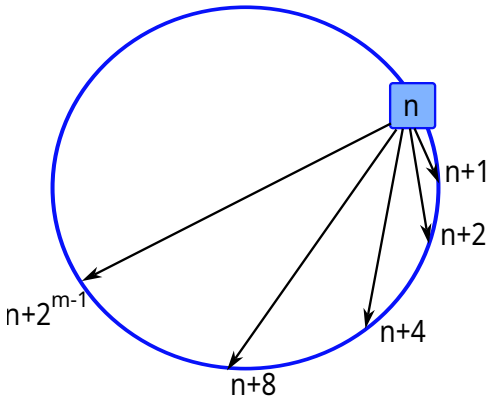


Figure 3.10: Illustration of chords

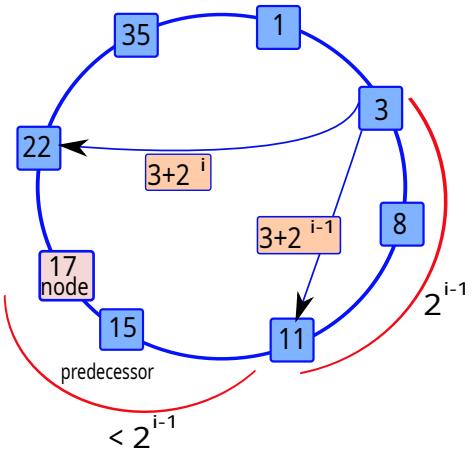


Figure 3.11: Fingers of node n where $i = 4$.

it is important to keep this in mind. Let us look at the last finger, which is also the largest finger. This finger is the m^{th} finger. It terminates at $(n + 2^m - 1) \, mod \, 2^m$. Since we are assuming that each id is m digits, the total number of ids that can be stored on the circle is 2^m. We thus see that the end of the last finger is the id that is just before the id of the current node (position n on the circle). Hence, all the fingers cumulatively cover the entire circle. The size of the first finger as we can see in Table 3.1 is just one point and the size of the last finger is 2^{m-1} points. In fact, the sizes of

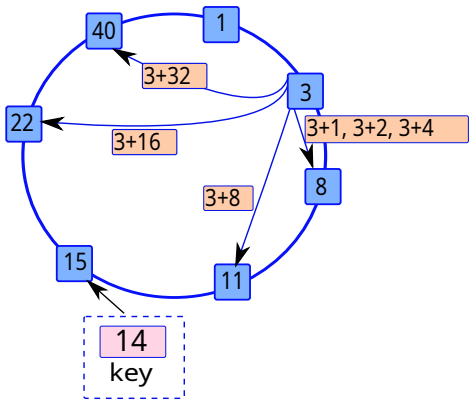


Figure 3.12: Fingers of node n

the fingers – number of points contained in them – increase exponentially: 1, 2, 4, 2^{m-1}. We are basically spanning larger and larger chunks of the id space, which nicely summarizes our discussion.

Finger (i)	Start	End
1	$n+1$	$n+1$
2	$n+2$	$n+3$
3	$n+4$	$n+7$
4	$n+8$	$n+15$
$\ldots$	$\ldots$	$\ldots$
m	$n+2^{m-1}$	$n+2^m-1$

Table 3.1: A tabular list of fingers . Note that all these additions are modular additions ($mod\, 2^m$).

Let us now take a look at the notion of a finger's node. It is the successor of the starting location of the finger. This may lie within the finger, or may actually be in a different finger. The lookup process is thus straightforward. We find the finger in which the key should lie and find the last node that precedes the finger. Note the word "precedes". This node should be in an earlier finger such that the search can start from it. Like Pastry, the lookup message is routed to this node. The entire procedure restarts from this node

in a recursive fashion. The reason that this process works is because we are effectively halving the search distance (if not more). Hence, this process takes a maximum of $log(n)$ steps, where n is the number of total nodes. Let us now look at the finer points.

3.7.2 Chord's Hashing Algorithm

The SHA-1 algorithm is used to hash node ids and keys. The SHA-1 algorithm is used. It produces a 160-bit (20-byte) value at the end of the process. We need a *consistent hashing* scheme where regardless of the number of nodes keys are roughly uniformly distributed amongst them.

For example, if there are n nodes and k keys, then we expect to have k/n keys per node. If the hashing process produces random numbers that are uniformly generated, then theoretically this will be the case for large values of k and n (as per the law of large numbers). However, there is a finite probability of a deviation from the uniform distribution. This unduly load some nodes and reduce the load at many other nodes.

There is an interesting solution for this. It is known as *virtual nodes*. Let us look at the maths [68, 112]. We can prove that with a very high probability, each node stores at the most $(1 + \epsilon)k/n$ keys. Any addition of a node and subsequent redistribution of keys moves around $O(k/n)$ keys. However, the more important result is that $\epsilon = O(log(n))$. This is seen to be militating against the philosophy of consistent hashing. There is a need to achieve more uniformity particularly if the number of keys is on the higher side (like in a large e-commerce site).

The following trick ensures that this $log(n)$ factor is successfully removed. We use $\kappa = log(n)$ hash functions. All of them need to be consistent hashing algorithms (similar to SHA-1). A standard approach to do this is to append a few additional bits (per hash function) known as *salt* values. The algorithms are designed in such a way that even if there is a small variation in the text (a few bits), the output is very different. Our hash functions are thus as follows: $\mathcal{H}_1 \ldots \mathcal{H}_\kappa$. Now, for every single node we create κ different virtual nodes, all placed at different positions on the id circle. We are thus artificially inflating the number of nodes by creating $log(n)$ copies of each physical node and placing them at different nodes on the id circle.

In totality, we thus have κn virtual nodes. If $\kappa = log(n)$, we have $nlog(n)$ nodes. It can be proven that this trick ensures that the key distribution is roughly balanced. At the cost of additional computational and routing complexity, using the notion of virtual nodes helps in better key distribution.

Remark 15 *Using virtual nodes in Chord ensures balanced key distribution per physical node. This comes at the cost of increased routing complexity.*

3.7.3 Chord's Routing Algorithm

Algorithm 2 shows the high-level overview, which is actually quite simple. We find the predecessor `P` of the key whose hash value is `id`. Every node has a pointer to its immediate successor and predecessor on the ring. The node that the key is mapped to is just the successor of `P` (by definition). This can be easily visualized with an example. The `id` will lie between `P` and `P.successor`. Hence, the node that `id` is mapped to is `P.successor`. Since every node has a successor, it is quite easy to find `P`'s successor. Let the term `cur` refer to the current node that is running the algorithm.

Algorithm 2 Find the node that is mapped to the key with hash `id`

```
1: procedure findSuccessor(id)
2:     mP ← findPredecessor (id)
3:     return P.successor
4: end procedure
```

Now, the primary task is to find the predecessor of the key. This is where the finger table will prove to be useful. Let us first look at the algorithm to find the closest preceding finger node for a key with hash `id` (refer to Algorithm 3). The idea is to traverse the list of fingers in descending order – largest finger first. The process of traversal terminates when the finger's node lies between the id of the current node and the hash of the key (`id`). The order should be: ⟨current node initiating the search⟩ → ⟨finger's node⟩ → ⟨hash of the key⟩. This ensures that we reach a node that is closer to the key on the circle than the current node. We shall later on prove that we are at least halving the distance; however, at the moment just saying that *we are getting closer* suffices. Note that in Algorithm 3, the operation ∈ takes two arguments that correspond to two different nodes on the id circle. We can assume this to be an overloaded function whose arguments can either be nodes or hashes or any combination thereof.

Algorithm 3 Find the closest preceding finger's node

```
1: procedure closestPrecedingFinger(id)
2:     for i ← m to 1 do
3:         if finger[i].node ∈ (cur, id) then return finger[i].node
4:         end if
5:     end for
6:     return cur                          ▷ The current node is the closest.
7: end procedure
```

Let us now take a deeper look at Algorithm 3. In Line 2, we iterate through the fingers (largest first). The m^{th} finger (or chord) corresponds to the semicircle. We first check if the `id` is in the other semicircle with respect to the current node. Of course, it is possible that the finger's node may not be close to its starting point. Hence, the check in Line 3 is slightly more nuanced. We find if the current finger's node is between the current node (`cur`) and the hash of the key (`id`). If that is the case then the current finger's node is the closest to the key. If the check in Line 3 fails, which means that the current finger's node is not in between `cur` and `id`, then we know that its predecessor must be in the same semicircle as `cur`. We repeat the same process iteratively dividing the region of interest by a factor of 2 repeatedly. Finally, if the check in Line 3 is successful, then we return `finger`[i].`node`, which is the node corresponding to the i^{th} finger (lies between the current node and the key). If no such node is found, then we come to Line 6, where we conclude that the current node is the closest to the key and we return it. Using this algorithm we need to find the predecessor of a key. The steps are shown in Algorithm 4.

Algorithm 4 Find the predecessor

```
1: procedure findPredecessor(id)
2:     tmp ← cur
3:     while id ∉ (tmp, tmp.successor) do
4:         tmp ← tmp.closestPrecedingFinger(id)
5:     end while
6:     return tmp
7: end procedure
```

The crux of Algorithm 4 is to keep getting closer and closer to the key until it is not possible to get any closer. Here, *getting closer* means getting closer while still preceding the key on the id circle. We keep calling the

function closestPrecedingFinger() repeatedly in Line 4. In each iteration of the loop, we find the finger node that is the closest to the `id` yet still precedes it, as the name of the function suggests. This is our new starting point. We again start from here. Given that we are closer to the key now, we can rely on more the fine-grained information in the finger table to get even closer. Ultimately, we reach the real predecessor of the key.

Corner case: An astute reader may argue that these algorithms do not handle the case in which the id of a node matches the id (hash) of a key. This is in reality very rare given that the key space is very large (2^{128}). Theoretically, this can happen. These algorithms will require minor changes to incorporate these conditions. For the sake of readability, these conditions have been ignored. However, they are easy to add and have been left as an exercise for the reader.

Remark 16 *In Chord, the finger table replaces the routing table. We traverse it in a descending order of the finger's size to find the next node that we need to route the message to. The protocol is more robust unlike Pastry. It is not dependent on how well the routing table is populated and thus guaranteeing quick forward progress is easier.*

3.7.4 Node Arrival

In any system that uses DHTs, nodes keep on arriving and departing. Hence, there is a need to continuously modify the DHT to include new nodes and remove older nodes. This is in fact one of the key advantages of a DHT – we can grow and shrink it at will. There are two key aspects of adding a new node. We need to update the state of the node that is added and inform other nodes about the current node. This is quite similar to Pastry. However, given that Pastry uses more state information (leaf set and neighborhood set), this process requires more messages. Node addition in the case of Chord is far easier. We precisely know which nodes will change their state – they can be informed accordingly.

Initialize the Finger Table

Initializing the finger table is the only step in initializing the state of a new node that is going to be added. This process is shown in Algorithm 5. We first update the predecessor and successor information. Then we find the nodes corresponding to each of the fingers and update the finger table with this information.

Algorithm 5 Initialize the finger table

```
1:  procedure initFingerTable(Node T)
2:      /* Update the predecessor and successor */
3:      finger[1].node ← T.findSuccessor (finger[1].start)
4:      successor ← finger[1].node
5:      predecessor ← successor.predecessor
6:
7:      /* Update the predecessor and successor of neighboring nodes */
8:      successor.predecessor ← cur
9:      predecessor.successor ← cur
10:
11:     /* Initialize the rest of the fingers */
12:     for i ← 1 to m − 1 do
13:         if finger[i + 1].start ∈ (cur, finger[i].node) then
14:             finger[i + 1].node ← finger[i].node
15:         else
16:             finger[i + 1].node ← T.findSuccessor
    (finger[i + 1].start)
17:         end if
18:     end for
19: end procedure
```

The current node `cur` calls the function initFingerTable(). For getting all the information that it needs as a part of this function, it needs the help of another node `T`. Node `T` exports a couple of functions that the newly added node (`cur`) needs to use.

For every finger, we know the start and end values as per Equation **??**. We need to find each finger's node, which is the successor of the start pointer (in the id circle). Doing this for the first finger is not very hard as observed in Line 8. The successor of finger[1].start is the finger's node. Of course, we need the help of node `T` to achieve this task. Node `T` is already there in the DHT and its finger tables are fully initialized. We then go on to initialize the two local variables associated with each node: `successor` and `predecessor`. The `successor` was obtained in the previous step. The predecessor of the current node is nothing but the successor's predecessor.

Then we move on to changing the state of the successor and predecessor. They need to be made aware of the current node that is being added. Their successor and predecessor pointers are appropriately modified in Lines 8 and 9.

Then, we proceed to initialize the rest of the fingers. We need to take the sparsity of nodes along the id circle into account. If `finger[i + 1].start` is in between the current node `cur` and `finger[i].node`, then it means that if we start traversing the id circle clockwise from `finger[i + 1].start`, the first node that we will hit is `finger[i].node`. We can thus set `finger[i + 1].node` to `finger[i].node` (see Line 14). If this is not the case, then we need to initiate a search to find `finger[i + 1].node`. We can leverage node `T`'s FINDSUCCESSOR() method that finds the successor start from the location `finger[i + 1].start`.

Update the State of the Rest of the Nodes

Given that the state of the current node has been updated, we need to update the state of the rest of the nodes. We have already some of the work in Algorithm 5 by updating the predecessor and successor pointers of the succeeding and preceding nodes, respectively. The only task that is remaining is updating the finger tables of other nodes. It is possible that the current node `cur` that is being added is a finger's node in the rest of the nodes. Note that all arithmetic operations are modular arithmetic operations (mod 2^m).

Algorithm 6 Update the finger tables of other nodes

```
1:  procedure cur.updateOthers(())
2:      for i ← 1 to m do
3:          ▷ This node may have the current node in its finger table
4:          pred ← findPredecessor (cur.hash - 2^(i−1))
5:          pred.updateFingerTable (cur, i)
6:      end for
7:  end procedure
8:  procedure pred.updateFingerTable(Node T, int i)
9:      ▷ Update the finger table of a node if T is a finger's node
10:     if T ∈ (pred, finger[i].node) then
11:         finger[i].node ← T
12:         predecessor.updateFingerTable (T, i)        ▷ Recursive call
13:     end if
14: end procedure
```

Algorithm 6 shows the pseudocode for updating the finger tables of other nodes. The current node that is being added (`cur`) calls the function UPDATEOTHERS(). We start with the smallest finger and keep moving towards

larger fingers. Consider the i^{th} finger. The current node can be in the finger table as the i^{th} finger's node for any node that has a hash value less than `cur.hash` - 2^{i-1} (mod 2^m). Starting from this index, we search for nodes with lower hash values. This is the same as finding the predecessor of this value (see Line 4). Starting from this predecessor node, we start updating the finger tables of all preceding nodes (refer to the function updateFingerTable()).

The function updateFingerTable invoked by a predecessor node takes two arguments: the node that is going to be added (`T`) and the number of the finger (`i`). The vital check is on Line 10. Let us understand the relationship between the node to be added (`T`), `finger[i].node` (in the node `pred`) and `pred` itself.

We are interested in the i^{th} finger of `pred`. We know the following:

$$\begin{aligned} &pred.finger[i].start = pred.hash + 2^{i-1} \\ &pred.finger[i].node.hash > pred.finger[i].start \\ &n.hash > 2^{i-1} + pred.hash = pred.finger[i].start \end{aligned} \tag{3.11}$$

The relationship between `T` and `pred.finger[i].node` is thus not clear. It is clear that both are larger than `pred.hash` and `pred.finger[i].start`. There are two cases: either `T` is in between `pred` and `pred.finger[i].node` (Case I) or not (Case II).

Let us consider Case I. Given that `T` is located after `pred.finger[i].start` on the id circle and it is located before `pred.finger[i].node`, we can conclude that `pred.finger[i].node`'s new value should be `T`. It is closer to `pred.finger[i].start`. This is exactly what is done in Line 11. Now, it is possible that additional nodes are also affected. We thus perform the same procedure on *pred*'s predecessor (Line 12) and keep doing this as long as the check in Line 10 is satisfied.

Now, if we consider Case II. In this case, `T` is not located after `pred.finger[i].node`. As a result, the value of `pred.finger[i].node` does not have to be updated. It is not affected by the addition of the new node. The same is the case for `pred`'s predecessors as well. As a result, in this case, we can just return from the updateFingerTable (...) function without performing any action.

3.7.5 Node Departure

Let us now see what happens if a node leaves the Chord network. This can happen in two ways. One is that the node voluntarily leaves. For instance,

we decide to scale down the Chord network and power servers down. In this case, there is a case for a controlled exit where the exiting node can initiate a departure protocol. If a node crashes, then other nodes need to detect its exit and initiate a departure protocol. In both cases, the departure protocol is quite similar.

Let us assume the simpler case where a node voluntarily leaves. The first action is to inform the successor and predecessor nodes. They need to update their respective successor and predecessor pointers. The more difficult task is to update the finger table.

Algorithm 7 Delete the node from finger tables

```
1: procedure CUR.DELETENODE(())
2:     for i ← 1 to m do
3:         pred ← FINDPREDECESSOR (cur.hash - 2^(i−1))
4:         pred.DELETENODE (cur, i)
5:     end for
6: end procedure
7: procedure PRED.DELETENODE(Node T, int i)
8:     if finger[i].node = T then
9:         finger[i].node ← T.successor
10:        predecessor.DELETENODE (T, i)
11:    end if
12: end procedure
```

We follow a similar procedure as adding a new node. The current node `cur` that is going to be deleted calls the DELETENODE() function. We iterate through all the fingers and delete the current node from the finger tables of all the nodes that may contain it. Similar to the case of adding a new node, we find the predecessor node of `cur.hash` - 2^{i-1}. This node is `pred`. We then call `pred`.DELETENODE (...).

If `pred.finger[i].node` = `n`, then we need to set a new value for the finger's node. This node in this case will be `T.successor`. This is the new node for the `pred`'s i^{th} finger. Now, it is possible that other nodes would have `T` as the node of their i^{th} finger. We follow the same procedure and recursively call the DELETENODE (...) function for the predecessor of `pred` and so on (until the process terminates).

3.7.6 BitTorrent

3.8 FreeNet

Chapter 4

Mutual Exclusion Algorithms

In a distributed system, we have a lot of shared resources. Different processes that run concurrently may attempt to acquire these shared resources in any order. In this case, we can have a *race condition* where different processes can simultaneously try to either access or modify the state of the shared resource. This can lead to many correctness issues. Many times these are very subtle issues that manifest as errors or failures much later. As a result, there is a need to ensure that only one process can access any of these critical resources at any given point of time. This property is formally known as *mutual exclusion*. Arguably, this is the most important property in a concurrent system because it ensures correctness as well as from the point of view of the shared resource/object, it provides an illusion of sequential execution. We shall start the discussion in this chapter by looking at some of the classical mutual exclusion algorithms. We will refer to this shared resource that only admits exclusive ownership as a *lock*, henceforth.

In all distributed algorithms, we need to prove two things: safety and liveness. *Safety* means that something bad never happens. For example, in a traffic signal, it is never the case that two lights for perpendicular roads are green at the same time. This would lead to a collision. The other important property is *liveness*. This means that something good always happens. Consider the traffic signal example again. If we can guarantee that a light eventually turns green, then this is a good thing – it guarantees forward progress. Hence, we can conclude that for a traffic signal to work, we need to guarantee both safety and liveness.

Definition 17 *Safety means that something* bad *never happens in a system.*

For instance, it is never the case that traffic lights in two perpendicular directions are green at the same time. Liveness *means that something good always happens. In the example of a traffic intersection, this means that a traffic light always turns green.*

In the case of distributed algorithms like mutual exclusion, safety typically means that two processes do not access the lock at the same time. This would break the fundamental premise of mutual exclusion. Liveness, in this case, means that every process is guaranteed to get the lock. There, of course, can be different kinds of definitions for liveness. We may want to ensure that every process gets the lock in a finite amount of time. Some may argue that this is too open-ended and would like every interested process to get access to the lock in a bounded amount of time. Regardless of the definition of liveness, the basic idea is that there is no *starvation*. Starvation means that it is never the case that a process times out while trying to access a lock. The time out value is dependent on the definition of liveness that we are using. However, conceptually *starvation freedom* would mean guaranteed access before the user decides that it is too late. Starvation freedom implies several things. For instance, it implies that processes cannot *deadlock*, which is defined as a circular wait condition where process `P` waits on process `Q`, which waits on process `R`, so and so forth, and finally the last process in the list waits on process `P`. As a result, none of the processes in this circular dependence loop can make any progress. It is clear that starvation freedom implies deadlock freedom. The converse is not necessarily true.

Fact 18 *Starvation freedom implies deadlock freedom. The converse is not necessarily true.*

Keeping these concepts in mind, let us move forward and introduce distributed algorithms for mutual exclusion.

4.1 Tokenless Distributed Algorithms for Ensuring Mutual Exclusion

In this section, we will mostly deal with the simpler variety of distributed algorithms for ensuring mutual exclusion – asynchronous algorithms that do not have faulty processes (refer to Section 1.1).

4.1.1 Lamport's Mutual Exclusion Algorithm

This is a 3-phase algorithm: request, reply and release. The broad idea is as follows. Whenever a process wants to access a critical section, it sends a message to all the n processes (including itself). Every process, maintains a queue of requests. It sends a $\langle$`reply`$\rangle$ message to a lock request message as soon as it gets it. Once, a process has gotten $n-1$ messages that are newer than the request and no other requests precede its request, it knows that it can access the lock . After accessing the lock, it sends a $\langle$`release`$\rangle$ message to the rest of the processes. After receiving this message they can remove the original request from their queues and proceed with the rest of the requests that they have in their respective queues. This is a very high-level description of the algorithm. Let us now look at the algorithm in its full glory.

Algorithm 8 shows the algorithm. Here, we make several assumptions. We assume that the Lamport's scalar clock (refer to Section 8.9) is used to maintain a notion of logical time. All such scalar clocks are comparable even if they have the same value – we can break ties based on the process id. Furthermore, we assume FIFO channels. This means that all the messages that are sent from Process u to v are ordered. It is never possible for a later message to overtake an earlier message and reach v sooner.

Algorithm 8 Lamport's mutual exclusion algorithm

1: **procedure** REQUEST(Process $\mathtt{P_u}$)
2: $\mathtt{P_u}$ sends a $\langle \mathtt{request} \rangle$ message $\langle \mathtt{T_u}, u \rangle$ to all processes
3: It puts this message in its own request queue
4: When $\mathtt{P_v}$ receives a $\langle \mathtt{request} \rangle$ message from $\mathtt{P_u}$, it sends a timestamped acknowledgement
5: **end procedure**
6: **procedure** ACCESS(Process $\mathtt{P_u}$)
7: ▷ Access the lock if the following two conditions hold, else wait for them to become true.
8: ▷ The original $\langle \mathtt{request} \rangle$ message $\langle \mathtt{T_u}, u \rangle$ is the earliest message in the queue
9: ▷ Process $\mathtt{P_u}$ has received a message with a timestamp greater than $\mathtt{T_u}$ from all the other nodes.
10: **end procedure**
11: **procedure** RELEASE(Process $\mathtt{P_u}$)
12: $\mathtt{P_u}$ removes the original $\langle \mathtt{request} \rangle$ message from its queue
13: It sends a timestamped $\langle \mathtt{release} \rangle$ message to all the other nodes
14: When a process $\mathtt{P_v}$ receives a $\langle \mathtt{release} \rangle$ message from $\mathtt{P_u}$, it removes the corresponding $\langle \mathtt{request} \rangle$ message
15: **end procedure**

Let us consider the function REQUEST. The main aim of this function is to register a request with every process including itself. This bears the local timestamp of $\mathtt{P_u}$. We assume that every process has a queue where it can keep requests and their replies in ascending order of their timestamps. Note that since Lamport timestamps are scalar timestamps, they are always mutually comparable – we can decide which one is greater (use the process id as a tie breaker), even though this many not necessarily imply causality. Every process other than $\mathtt{P_u}$ acknowledges a request by sending a timestamped reply. The timestamp of the reply needs to be strictly greater than the timestamp of the request. This is as per the rule for updating the logical time.

The request's timestamp is $\mathtt{T_u}$. When it is received by process $\mathtt{P_v}$, it will update its local time if there is a need. In any case, its updated local time is guaranteed to be at least as large as $\mathtt{T_u} + 1$. The timestamp of the reply will thus be at least $\mathtt{T_u} + 1$. This means that when $\mathtt{P_u}$ receives a reply from any other process, its reply timestamp will at least be $\mathtt{T_u} + 1$. It is of course possible that $\mathtt{P_u}$ gets another message from $\mathtt{P_v}$ with a timestamp greater

than $\mathtt{T_u}$ before the reply is sent. We will not distinguish between these two cases.

Now, process $\mathtt{P_u}$ accesses the lock only when two conditions hold, else it *waits* for both of them to hold. The first condition is shown in Line 8 – the original message with timestamp $\mathtt{T_u}$ is the earliest message in its request queue (of process $\mathtt{P_u}$). This means that in the queue of requests, it has the least timestamp.

The second condition is listed in Line 9 – $\mathtt{P_u}$ should have received a message with a timestamp greater than $\mathtt{T_u}$ from every other process.

Now, we need to read both these conditions together. The key question that we need to answer is whether it is possible for two processes, $\mathtt{P_u}$ and $\mathtt{P_v}$ to be in the critical section (accessing the lock) at the same time. The second question is whether a process is guaranteed to get the lock once it has made a request. This assumes that every time lock is acquired, it is released in a bounded amount of time. Let us first prove the former (see Lemma 19).

Lemma 19 *It is not possible for two processes* $\mathtt{P_u}$ *and* $\mathtt{P_v}$ *to concurrently access the locked resource at the same time.*

Proof: Let us assume to the contrary that the processes are indeed accessing the shared/locked resource at the same time. Process $\mathtt{P_u}$ must have sent a message with timestamp $\mathtt{T_u}$ to $\mathtt{P_v}$. It would have then found out that its request had the least timestamp in its request queue and it had received a message with timestamp greater than $\mathtt{T_u}$ from $\mathtt{P_v}$.

We need to see what exactly must have happened at $\mathtt{P_v}$. It would have received the message from $\mathtt{P_u}$, set its internal timestamp to at least $\mathtt{T_u} + 1$ and sent a reply. It could not have generated its access request after this point because its request would then have a timestamp greater than $\mathtt{T_u}$ and Condition 1 will get violated.

Hence, we can conclude that $\mathtt{P_v}$ must have generated its request before it received the $\langle\mathtt{request}\rangle$ message from $\mathtt{P_u}$. The question is whether the timestamp of its request $\mathtt{T_v}$ is greater than $\mathtt{T_u}$ or not. Note that we will never have equality because the process id will be used to break ties.

Case I: $\mathtt{T_v} > \mathtt{T_u}$. In this case, $\mathtt{P_u}$ would have received the $\langle\mathtt{request}\rangle$ message from $\mathtt{P_v}$ before it received the reply to its own $\langle\mathtt{request}\rangle$ message. Given that $\mathtt{T_v} > \mathtt{T_u}$, it means that $\mathtt{P_u}$ would have sent its $\langle\mathtt{request}\rangle$ message by now, otherwise it would have had a higher timestamp. Given that $\mathtt{P_u}$'s request message was on the way, when $\mathtt{P_v}$'s $\langle\mathtt{request}\rangle$ message was received at $\mathtt{P_u}$ and we assume FIFO channels, we can safely conclude that $\mathtt{P_v}$ would

have seen $\texttt{P}_\texttt{u}$'s ⟨`request`⟩ message before seeing a message from $\texttt{P}_\texttt{u}$ with a timestamp greater than $\texttt{T}_\texttt{v}$. Hence, Condition 1 will not succeed at $\texttt{P}_\texttt{v}$ because $\texttt{P}_\texttt{u}$'s request has a lower timestamp than its own request. Hence, unless $\texttt{P}_\texttt{u}$ releases the lock, $\texttt{P}_\texttt{v}$'s request cannot go through.

Case II: $\texttt{T}_\texttt{v} < \texttt{T}_\texttt{u}$. Starting from the basic assumption, the ⟨`request`⟩ message was sent by $\texttt{P}_\texttt{v}$ before it received $\texttt{P}_\texttt{u}$'s ⟨`message`⟩. This means that after sending a message timestamped with $\texttt{T}_\texttt{v}$, $\texttt{P}_\texttt{v}$ sent a message with a timestamp greater than $\texttt{T}_\texttt{u}$ (because $\texttt{P}_\texttt{u}$ ended up getting the lock). This means that before Condition 2 started to hold at $\texttt{P}_\texttt{u}$, it had received $\texttt{P}_\texttt{v}$'s request, and this message had a lower timestamp that $\texttt{P}_\texttt{u}$'s original ⟨`request`⟩ message. This means that Condition 1 could never have been true at $\texttt{P}_\texttt{u}$. Hence, $\texttt{P}_\texttt{u}$ could not have accessed the lock when $\texttt{P}_\texttt{v}$ was accessing it. In this case, $\texttt{P}_\texttt{u}$ can only access after $\texttt{P}_\texttt{v}$ releases.

Given cases I and II, we can clearly see that concurrent access to the resource is not possible. □

Proving that there is no *starvation* – a process does not wait forever – is easy. We need to note that Lamport timestamps increase monotonically. They never decrease. As a result, ultimately a request will become the oldest request in the system. Furthermore, the requesting process will eventually get a reply to its request. The reply is guaranteed to have a greater timestamp than the request. This means that Conditions 1 and 2 will ultimately get satisfied and the requesting process will get access to its resource. In fact, we can easily prove that in the worst case a process needs to wait for the rest of the $n - 1$ processes to access the resource. At that point, the request from the process will become the oldest in the system. This is easily visible because once a process replies to a request, its subsequent request can only have a greater timestamp.

The Lamport's algorithm is simple and straightforward. However, proving that it works, especially that it guarantees mutual exclusion is difficult as we saw in the proof for Lemma 19. The standard approach for all such proofs is the same. We prove by contradiction. We assume that the property that we are trying to prove is violated. Then, we break the symmetry and consider different cases ($\texttt{T}_\texttt{u} > \texttt{T}_\texttt{v}$ and $\texttt{T}_\texttt{u} < \texttt{T}_\texttt{v}$). For each case, we derive a contradiction. Proving starvation freedom also typically involves proofing that ultimately the request becomes the oldest request in the system.

Message Complexity

Unlike sequential algorithms, the complexity of distributed algorithms is measured differently. We don't count the number of steps that a processing

unit takes. We instead count the total number of messages and sometimes the total number of rounds required to complete the protocol in the case of a synchronous algorithm. The number of messages is by far the most common metric. This is because the time that a network message takes is typically far more than the time taken by the program to process the message.

In this case, there are three phases for accessing a resource. In each phase, a message needs to be sent to the rest of the $n-1$ processes. Thus, we arrive at a total count of $3(n-1)$ messages.

4.1.2 Ricart-Agarwala Algorithm

Is there a need to send a reply acknowledging the receipt of a message immediately? It is actually not required. We can instead send the acknowledgement later on and piggy back it with a release message. This will reduce the total message complexity from $3(n-1)$ to $2(n-1)$.

This algorithm is deceptively simple (see Algorithm 8.9).

Algorithm 9 Ricart-Agarwala algorithm

```
1: procedure REQUEST(Process P_u)
2:     P_u sends a timestamped ⟨request⟩ message to all nodes
3: end procedure
4: procedure RECEIVEREQUEST(P_v receives a request from P_u)
5:     if P_v is not holding the lock and not interested in it then
6:         return timestamped ⟨reply⟩ message
7:     end if
8:     if P_u's ⟨request⟩ timestamp is lower than P_v's and P_v is not holding
   the lock then
9:         return timestamped ⟨reply⟩ message
10:    end if
11:    Queue the request
12: end procedure
```

The function REQUEST simply sends a timestamped message to all processes requesting the lock. The key logic is embedded in the function RECEIVEREQUEST. There are two conditions. If these conditions are not true, then the request is queued.

First, if P_v is not interested in the lock or is not currently holding it, then it should not have any objections to P_u acquiring the lock. A timestamped ⟨reply⟩ message can be sent (see Line 6). The other case is that P_v is interested in acquiring the lock but does not currently hold it. This case

is captured in Line 9. The algorithm makes $\texttt{P}_\texttt{v}$ relinquish its claim if $\texttt{P}_\texttt{u}$'s request has a lower timestamp. In this case also $\texttt{P}_\texttt{v}$ sends a timestamped reply. Essentially, $\texttt{P}_\texttt{v}$ passes the baton to $\texttt{P}_\texttt{u}$ because the latter's request was earlier.

The process of acquiring the lock is similar to the Lamport's algorithm. When a process has received $n-1$ replies from the rest of the $n-1$ nodes, it is ready to acquire the lock. When a lock is released, the process replies to all pending requests.

The message complexity is $2(n-1)$. The reply message in this case was deliberately delayed. As a result, we claim that the algorithm achieves using fewer messages. Let us now look at the proof.

Proof of Correctness

We shall follow the same approach here as well. Assume that processes $\texttt{P}_\texttt{u}$ and $\texttt{P}_\texttt{v}$ acquire the lock concurrently. Let $\texttt{P}_\texttt{u}$ have the lower request timestamp. This basically means that after $\texttt{P}_\texttt{u}$ generated its request, it must have gotten $\texttt{P}_\texttt{v}$'s request. Otherwise, it would have had a higher request timestamp – it would have incorporated the time information of $\texttt{P}_\texttt{v}$ and increased the value of its Lamport clock.

Then, this also means that as per Line 9, $\texttt{P}_\texttt{u}$ could not have sent a reply to $\texttt{P}_\texttt{v}$. It would have been aware that it is interested in acquiring the lock and its request timestamp is lower than that of $\texttt{P}_\texttt{v}$'s. As a result, $\texttt{P}_\texttt{v}$ would not have gotten any timestamped reply on $\texttt{P}_\texttt{u}$. This means that $\texttt{P}_\texttt{v}$ could not have possibly acquired the lock.

This leads to a contradiction.

4.1.3 Maekawa's Algorithm

Can we reduce the message complexity even further? Can it be made $O(\sqrt{n})$. This would require some new insights. Let us take a second look at the proof of the Ricart-Agarwala algorithm. The proof hinges on the fact that there is a node running process $\texttt{P}_\texttt{u}$ that will never send a reply to the node running process $\texttt{P}_\texttt{v}$. This is because, this would be violative of a condition in the algorithm that prohibits a process from sending a reply it its request's timestamp has a lower timestamp than the request received from a remote node. The insight that we can derive is that the process that holds back the reply to $\texttt{P}_\texttt{v}$ performs a key synchronizing function (ordering one after the another). It does not allow concurrent lock accesses by deliberately delaying one process.

Hence, note that for any two requests, all that we need to synchronize them is just one process. It holds something akin to a veto power. Given that a reply from it is required, it may choose to reply to one process and not reply or delay the reply to the other.

Let us develop this idea further. Let us associate a set of processes with process $\texttt{P}_\texttt{u}$. This set is known as its *request set* R_u. Informally, R_u refers to the processes that $\texttt{P}_\texttt{u}$ requests for a lock. Similarly, for process $\texttt{P}_\texttt{v}$ we will have its corresponding request set R_v. The key property that we need to ensure is that for any given $\texttt{P}_\texttt{u}$ and $\texttt{P}_\texttt{v}$, $R_u \cap R_v \neq \phi$. The intersection of any two request sets is always non-null – there is at least one process in common. An easy method of constructing such a request set is shown in Figure **??**. Here, we arrange all the processes in a square grid. The request set of a process is defined as the set of processes in the row and column of the process. It is easy to see that the request sets will always have 2 intersections. This approach uses a request set with size $2\sqrt{n}$. It turns out that we can do much better and bring this number down to $\sqrt{n}$ if we use results from field theory. Let us stick to the simpler variant for the time being.

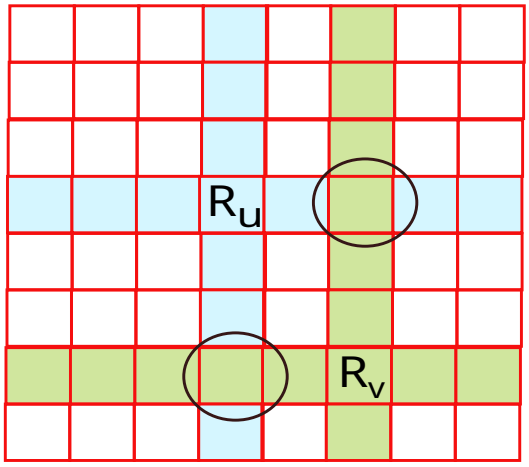


Figure 4.1: Arrangement of the processes in a matrix. The figure shows two overlapping request sets for the the requests R_u and R_v.

The summary of this discussion is that given any two processes, the intersection of their request sets is always non-null. In this case, instead of

sending messages and expecting replies from all processes, the lock requester needs to only ask its request set. Given that there will always be a common node (running a common process) between two request sets, it will have to play the role of ensuring mutual exclusion. Given that we are not considering faulty or inactive processes, the common node will always participate in the algorithm.

Acquiring the Lock

Let us first look at acquiring the lock. The philosophy is the same as earlier algorithms. We send a request to the request set, which in this case is much smaller, and wait for the replies. However, given that the request set is much smaller than the total number of processes, there are additional complexities. The main idea is that just delaying a reply is not always the best option because it can lead to deadlocks here. There is a need to send more messages.

Algorithm 10 Maekawa's algorithm: Acquiring the lock

1: **procedure** REQUEST(Process $\mathtt{P_u}$)
2: $\mathtt{P_u}$ sends a timestamped ⟨`request`⟩ message to all the nodes in R_u (including itself)
3: Wait till a ⟨`locked`⟩ message is received from all the processes in the request set, then acquire the resource
4: **end procedure**
5: **procedure** RECEIVEREQUEST($\mathtt{P_v}$ receives a request from $\mathtt{P_u}$)
6: **if** $\mathtt{P_v}$ is not locked **then**
7: $\mathtt{P_v}$ marks itself as locked **return** ⟨`locked`⟩ message to $\mathtt{P_u}$
8: **else**▷ This means that $\mathtt{P_v}$ is already locked by a request sent from $\mathtt{P_w}$
9: Place $\mathtt{P_u}$'s request in the request queue
10: If $\mathtt{P_u}$'s request is not the earliest request in the queue then send a ⟨`failed`⟩ message to $\mathtt{P_u}$
11: Otherwise, send an ⟨`inquire`⟩ message to $\mathtt{P_w}$
12: **end if**
13: **end procedure**
14: **procedure** RECEIVEINQUIRE($\mathtt{P_w}$ receives an ⟨`inquire`⟩ message from $\mathtt{P_v}$)
15: **if** $\mathtt{P_w}$ has already received a ⟨`failed`⟩ message or is not interested anymore **then**
16: Send back a ⟨`relinquish`⟩ message
17: **else**
18: Defer the reply
19: **end if**
20: **end procedure**
21: **procedure** RECEIVERELINQUISH($\mathtt{P_v}$ receives a ⟨`relinquish`⟩ message from $\mathtt{P_w}$)
22: Add the request from $\mathtt{P_w}$ to the queue of waiting requests
23: Find the earliest request in the queue and send its sender a ⟨`locked`⟩ message
24: **end procedure**

The REQUEST function looks the same (see Algorithm 10). The requesting process $\mathtt{P_u}$ sends a timestamped ⟨`request`⟩ message to all the processes in its request set (including itself)[1]. However, the way that another node reacts to this message is different. The receiving process $\mathtt{P_v}$ first checks if it

[1]Every process is a member of its own request set.

is locked in Line 6. The term "locked" here basically means "committed". This means that process $\texttt{P}_\texttt{v}$ has allowed some process (that may include itself) to acquire the lock (access the resource). Clearly, if $\texttt{P}_\texttt{v}$ is not locked, then it can send a message to $\texttt{P}_\texttt{u}$ with an "all-clear" from its side. This means that $\texttt{P}_\texttt{u}$ can assume that there are no objections from $\texttt{P}_\texttt{v}$'s side. This is conveyed using a ⟨`locked`⟩ message.

Let us consider the other case where $\texttt{P}_\texttt{v}$ has given a commitment (sent a ⟨`locked`⟩ message) to another process $\texttt{P}_\texttt{w}$. The first thing to do is to queue the message from $\texttt{P}_\texttt{u}$. Note that here a deadlock scenario is possible: process `P` is waiting on process `Q`, `Q` is waiting on `R`, so and so forth, and finally `R` waits on `P`. As a result, there is a need to do something other than plain old-fashioned waiting.

In Line 10, we check if the request from $\texttt{P}_\texttt{u}$ is the earliest in $\texttt{P}_\texttt{v}$'s queue. If it is not the earliest (lowest timestamp), then it means that the request from $\texttt{P}_\texttt{u}$ does not have the highest priority. As a result, a ⟨`failed`⟩ message is sent to $\texttt{P}_\texttt{u}$. It is an instantaneous acknowledgement. In case, if it is the earliest message, then there is a need to tell $\texttt{P}_\texttt{w}$ that a higher priority message has arrived. $\texttt{P}_\texttt{v}$ thus sends a message to $\texttt{P}_\texttt{w}$ inquiring if it is done in Line 11. The idea is to nudge $\texttt{P}_\texttt{w}$ to relinquish its claim to the resource. $\texttt{P}_\texttt{w}$ calls the receiveInquire message when it receives an ⟨`inquire`⟩ message from $\texttt{P}_\texttt{v}$.

In the receiveInquire function, $\texttt{P}_\texttt{w}$ checks if it has already received a ⟨`failed`⟩ message or it is not interested in the lock any more because it has used the resource, it sends back a ⟨`relinquish`⟩ message. In either case, it means that $\texttt{P}_\texttt{w}$ is not the first in line to acquire the resource. Either there are no competitors or there are competitors with earlier requests. In any case, the ⟨`relinquish`⟩ message indicates that $\texttt{P}_\texttt{w}$ is allowing $\texttt{P}_\texttt{v}$ to go ahead. If either of these conditions are not true, then the reply is deferred.

Let us now comes to the receiveRelinquish function. Once $\texttt{P}_\texttt{v}$ receives the ⟨`relinquish`⟩ message, it knows that its ⟨`locked`⟩ message to $\texttt{P}_\texttt{w}$ stands invalid. It is free to send a ⟨`locked`⟩ message to another process. It thus adds $\texttt{P}_\texttt{w}$'s request to its queue of waiting requests. It chooses the earliest request in the queue and sends a ⟨`locked`⟩ message to it. This could be $\texttt{P}_\texttt{u}$'s request or could be a request from some other process.

$\texttt{P}_\texttt{u}$ in the meanwhile waits till it has gotten a ⟨`locked`⟩ message from all the processes in its request set (see Line 3). Once all the ⟨`locked`⟩ messages

Releasing the Lock

Algorithm 11 Maekawa's algorithm: Releasing the lock

1: **procedure** RELEASE(Process $\mathtt{P_u}$)
2: It sends ⟨`release`⟩ messages to all the processes in its request set.
3: Once another process gets the ⟨`release`⟩ message, it locks the earliest message in its request queue or marks itself unlocked if the queue is empty.
4: **end procedure**

Algorithm 11 shows the logic for releasing the lock. $\mathtt{P_u}$ sends a ⟨`release`⟩ message to rest of the processes. Each process then initiates the locking procedure – mark itself as locked and send the ⟨`locked`⟩ message – for the earliest message in its request queue. If its request queue is empty, then the process marks itself as unlocked.

This part is quite straightforward bookkeeping. All the state that was kept locked because of $\mathtt{P_u}$ is released. The processes in the request set are free to lock themselves for other requests. A fresh iteration of the algorithm is started.

Proof of Mutual Exclusion

Let us prove along the same lines. Assume that two processes $\mathtt{P_A}$ and $\mathtt{P_B}$ acquire the lock concurrently. It is not possible for the common node $\mathtt{P_C}$ to send a ⟨`locked`⟩ message to both the nodes at the same time. It would have sent a ⟨`locked`⟩ message to one process $\mathtt{P_A}$ and marked itself as locked. It could not have then sent a ⟨`locked`⟩ message to process $\mathtt{P_B}$. This means that it is not possible for two processes $\mathtt{P_A}$ and $\mathtt{P_B}$ to hold the lock at the same time.

4.1.4 Proof of Starvation Freedom

Consider process $\mathtt{P_A}$, whose request is the earliest in the system. If it will get the lock in a finite amount of time, then there is no problem. However, let us consider the case when it waits indefinitely.

It means, it is waiting for some process $\mathtt{P_B}$ to reply with a ⟨`locked`⟩ message. $\mathtt{P_B}$ is not sending the ⟨`locked`⟩ message because of one of several reasons.

$\mathtt{P_B}$ may be using the resource itself. In this case, it will finish using the resource soon (our assumption) and then it will scan its request queue.

Since $\mathtt{P_A}$'s request is the earliest, $\mathtt{P_B}$ will send it the ⟨`locked`⟩ message. This is not a case of waiting because within a *short* period of time, $\mathtt{P_A}$ will get the ⟨`locked`⟩ message that it needs from $\mathtt{P_B}$.

Consider the more difficult case, where $\mathtt{P_B}$ has sent a ⟨`locked`⟩ message to $\mathtt{P_C}$. In this case, $\mathtt{P_B}$ sends an ⟨`inquire`⟩ message to $\mathtt{P_C}$. If $\mathtt{P_C}$ is holding the lock, it defers. However, we are not counting such *short* periods of time. The case that we are interested in, is when $\mathtt{P_C}$ defers indefinitely. This means that it has not gotten a ⟨`failed`⟩ message from any process in its request set. Now, the question arises if $\mathtt{P_C}$ has not received a ⟨`failed`⟩ message, then why has it not gotten access to the lock yet?

This further means that there is at least one process $\mathtt{P_D}$ that has not sent a ⟨`locked`⟩ message to $\mathtt{P_C}$. Now, $\mathtt{P_C}$'s request is the earliest at $\mathtt{P_D}$ because $\mathtt{P_C}$ has not gotten back a ⟨`failed`⟩ message from $\mathtt{P_D}$.

At $\mathtt{P_D}$, the same logic holds. It is waiting for another process $\mathtt{P_E}$ to reply with a ⟨`relinquish`⟩ message and $\mathtt{P_E}$ is not sending it.

Let us now quickly take a look at our processes. $\mathtt{P_A}$, $\mathtt{P_C}$ and $\mathtt{P_E}$ are lock requesters. $\mathtt{P_B}$ and $\mathtt{P_D}$ are processes in the request sets of $\mathtt{P_A}$ and $\mathtt{P_C}$, respectively. Furthermore, $\mathtt{P_A}$'s request is earlier than $\mathtt{P_C}$'s request, and $\mathtt{P_C}$'s request is earlier than $\mathtt{P_E}$'s request. This process cannot continue indefinitely. We have a finite number of processes and ultimately a situation will arise where we will observe that $\mathtt{P_A}$'s request is earlier than itself, which is not possible. Hence, there is a contradiction here and there will be no circular wait for an indefinite period.

This means that there will be no deadlocks and all the messages will be replied to within a finite period of time. **There will be no indefinite waiting**. This means that it will never be the case that a reply is deferred for perpetuity. All lock requests will get a reply in finite time, which will either be ⟨`locked`⟩ or ⟨`failed`⟩. Given that we are armed with this result, let us now prove that the system is starvation free.

Starvation Freedom: Let us prove this by contradiction. Consider a process $\mathtt{P_A}$ that starves. This means that for its ⟨`request`⟩ message, it gets at least one ⟨`failed`⟩ message. Then it does not get a ⟨`locked`⟩ message from the corresponding process in the request set. Consider a process $\mathtt{P_B}$ in its request set. The request for $\mathtt{P_A}$ can at the most be preceded by $\eta - 1$ processes in the queue of pending requests at $\mathtt{P_B}$. η is the size of the request set. In our construction, $\eta = 2\sqrt{n}$, whereas it can be reduced to $\sqrt{n}$. The reason for this limit is that it is not possible for the same process's request to precede the request made by $\mathtt{P_A}$ twice at any process in $\mathtt{P_A}$'s request set. This is because of the following reason. When $\mathtt{P_A}$'s request with timestamp τ arrives at $\mathtt{P_B}$, the request from $\mathtt{P_C}$ is already there in $\mathtt{P_B}$'s request queue.

At a later point of time a ⟨`locked`⟩ message is sent to $\mathtt{P_C}$. The timestamp of this message is greater than the timestamp of the message received from $\mathtt{P_A}$, τ (property of Lamport clocks). This means that the next request from $\mathtt{P_C}$ will have a timestamp greater than τ. It is thus not possible for another process $\mathtt{P_C}$ to generate request after request in a bid to stop $\mathtt{P_A}$'s request.

Ultimately after servicing a maximum of $\eta - 1$ requests, the request from $\mathtt{P_A}$ will become the earliest in the queue of pending messages at $\mathtt{P_B}$. It will then need to be sent a ⟨`locked`⟩ message. For a similar reason, $\mathtt{P_A}$ will get ⟨`locked`⟩ messages from the rest of the processes in the request set, and thus $\mathtt{P_A}$ will get the lock. Hence, there is no starvation.

Message Complexity

Let us assume that the size of the request set is η. As we have mentioned earlier, η can be made equal to $\sqrt{n}$, whereas in our system it is $2\sqrt{n}$. For every request, a maximum of $(\eta - 1)$ ⟨`request`⟩ messages are sent, $(\eta - 1)$ ⟨`locked`⟩ messages are received, $(\eta - 1)$ ⟨`failed`⟩ messages may be received, $(\eta - 1)$ ⟨`inquire`⟩ messages may be sent, and finally $(\eta - 1)$ ⟨`relinquish`⟩ messages my be received. Thus, the total message complexity is limited to $5(\eta - 1)$. This is $O(\sqrt{n})$.

Remark 20 *The Lamport's algorithm requires* $3(n-1)$ *messages, the Ricart-Agarwala algorithm requires* $2(n-1)$ *messages and the Maekawa's algorithm requires* $O(log(n))$ *messages.*

4.2 Token-based Distributed Algorithms for Ensuring Mutual Exclusion

Let us now look at a set of algorithms that explicitly use a token. The token is passed from process to process. The owner of the token is deemed to be the holder of the lock. This is a simple method of ensuring mutual exclusion. It is also quite easy to prove that the property of mutual exclusion is never violated.

4.2.1 Suzuki-Kasami Algorithm

Assume that every process maintains a sequence number – its request id. A request is of the form (u, m). This means that process $\mathtt{P_u}$ wants to access the lock for the m^{th} time. Let $\mathtt{P_u}$ also maintain an array $seq_u[1 \ldots n]$, where n is the number of processes. Here, $seq_u[v]$ is the largest sequence number

received from process $\texttt{P}_\texttt{v}$. This sequence number works like a Lamport clock. When $\texttt{P}_\texttt{u}$ receives (v, m), it does the following:

$$seq_u[v] = max(seq_u[v], m) \tag{4.1}$$

The job of this array is to maintain the latest state of $\texttt{P}_\texttt{v}$ (from the point of view of process $\texttt{P}_\texttt{u}$).

The token contains the following pieces of information:

- A queue (Q) of requesting sites.
- An array of sequence numbers, C. $C[u]$ is the latest request that $\texttt{P}_\texttt{u}$ completed.

The token is stateful. It contains a queue of pending requests. This maintains the ordering between the requests. It also contains the current state of completed requests across the system. Given the state of the token and the processes, let us look at the basic algorithms.

4.2.2 Acquiring the Lock

In Line 2 of Algorithm 8.9, we update the local clock of process $\texttt{P}_\texttt{u}$. Note that the entire algorithm is described from the point of view of $\texttt{P}_\texttt{u}$. In this case, it increments $\texttt{seq}_\texttt{u}[\texttt{u}]$. The new value is stored in `val`. This is sent to all the processes. When a different process $\texttt{P}_\texttt{v}$ receives $\langle \texttt{u}, \texttt{val} \rangle$, it updates its internal sequence number as per Equation 4.1.

Algorithm 12 Suzuki-Kasama Algorithm: Lock acquire

```
1: procedure ACQUIRE(Process Pu)
2:     val ← (++ sequ[u])
3:     Send ⟨u, val⟩ to all the processes
4:     Enter the critical section, when Pu has the token
5: end procedure
6: procedure RECEIVEACQUIRE(Pv receives ⟨u, val⟩)
7:     seqv[u] ← max(seqv[u], val)              ▷ update the local counter
8:     if Token.Q is empty and the token is there with Pv then
9:         if seqv[u] = token.C[u] + 1 then ▷ Check for staleness Send the
   token to Pu
10:        end if
11:    end if
12: end procedure
```

There is a fast path here (straightforward path). If $\texttt{P}_\texttt{v}$ has the token and the queue of processes in the token (`Q`) is empty, then the token can be sent to $\texttt{P}_\texttt{u}$ subject to a condition. It should not be the case that $\texttt{P}_\texttt{v}$ is reacting to a stale message. This means that $\texttt{P}_\texttt{u}$ has already used the token (acquired the lock) and one of its old requests is pending with $\texttt{P}_\texttt{v}$. There should be a method to adjudge the recency of the request. This is done using the check $\texttt{seq}_\texttt{v}[\texttt{u}] = \texttt{C}[\texttt{u}] + 1$. This means that the token has not been used by $\texttt{P}_\texttt{u}$ and $\texttt{P}_\texttt{v}$ rightly believes that $\texttt{P}_\texttt{u}$ hasn't used the lock and thus should get the token. The token is thus passed over to $\texttt{P}_\texttt{u}$.

However, most of the time, this will not be the case. There will be other contenders for the lock. This means that they will have entries in `token.Q` (the queue `Q` in the token).

Let us now look at the algorithm for releasing the lock.

Algorithm 13 Suzuki-Kasama Algorithm: Lock release

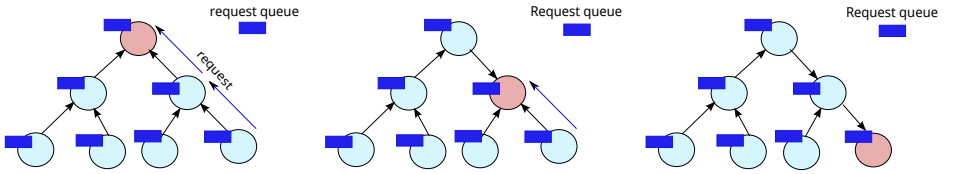
1: **procedure** RELEASE(Process $\texttt{P}_\texttt{u}$)
2: $\texttt{token.C}[\texttt{u}] \leftarrow \texttt{seq}_\texttt{u}[\texttt{u}]$ ▷ Update the state of the token
3: $\forall v$, add $\texttt{P}_\texttt{v}$ to `Q` if $\texttt{seq}_\texttt{u}[\texttt{v}] = \texttt{token.C}[\texttt{v}] + 1$ ▷ Perceive $\texttt{P}_\texttt{v}$ to be a contender for the lock
4: Dequeue $\texttt{P}_\texttt{w}$ from `Q` and send the token to $\texttt{P}_\texttt{w}$.
5: **end procedure**

Proof

Given that we have a single token that passes from process to process, proving mutual exclusion is simple. We have a single copy of the token. Whichever process needs to get the lock, needs to first get access to the token. After that, it can use the contended resource. Because of this, it is never possible to violate mutual exclusion because two processes will not have the token at the same point of time. This proof of mutual exclusion is simple.

Let us look at starvation. If a process is interested to get access to the resource, is it guaranteed to get the lock within finite time?

Whenever a process is interested in acquiring the lock, it sends a message to all the processes. The token maintains a queue (`token.Q`), which is an ordered list. Every request is guaranteed to get into the token's queue. The process that holds the token will add the request to the token's queue as soon as it gets the chance. There is a check for staleness in Lines 9 and 3 to ensure a request is not added to the token's queue more than once.



(a) Token at the root (b) Token on the way (c) Token at the destination

Figure 4.2: Different scenarios in the Raymond's tree algorithm

Once, the request is added to `token.Q`, it will gradually get promoted and will ultimately become the head of the queue. At this point of time, the requesting process is guaranteed to get the lock.

Message Complexity

There are two phases of the algorithm. $n-1$ messages are sent to the rest of the processes in the "lock acquire" stage. Then, the token is sent back to the requesting process once the corresponding request reaches the head of the queue. We thus have a total of n messages.

4.2.3 Raymond's Tree Algorithm

Let us now consider a far more scalable version of this algorithm. Let us reduce the message complexity even further from $O(n)$ to $O(log(n))$. This will require a tree-like structure. Basically, we need to organize all the processes as a balanced tree. Any balanced tree will have $O(log(n))$ levels. The idea is to pass the token between nodes in adjacent levels of the tree, and guarantee all the desirable safety and liveness properties.

Figure 4.2(a) shows one such scenario. In this case, the token is present with the root of the tree. The root is shown with a dark shade (dark pink, specifically). Every node points to its parent. Ultimately, the pointers are used to reach the root (refer to the arrows in Figure 4.2(a)). Every node has a queue of requests.

The interesting part of the algorithm is shown in Figures 4.2(b) and 4.2(c). The token moves from node to node across adjacent levels based on the pattern of requests. The token holder is always the root. The pointers orient themselves accordingly.

Remark 21 *The token holder in the Raymond's tree algorithm is always the* root *node. The token moves between adjacent nodes across levels. The*

pointers reorient themselves accordingly.

Algorithm 14 Raymond's tree algorithm: send lock request

```
1:  procedure sendRequest(Process P_u)
2:      if P_u is the root and the request queue is empty then
3:          Grab the lock
4:      else
5:          P_u adds itself to its request queue
6:          if P_u.alreadyForwarded = false then
7:              Forward a ⟨request⟩ message to its parent
8:              P_u.alreadyForwarded ← true
9:          end if
10:     end if
11: end procedure
12: procedure receiveRequest(P_v receives a message from P_u)
13:     if The request queue at P_v is empty and P_v has the token then
14:         Forward the token to P_u
15:     else
16:         Add the request from P_u to the request queue (of P_v)
17:         if P_v.alreadyForwarded = false ∧ P_v is not the root then
18:             Send a ⟨request⟩ message to P_v's parent
19:             P_v.alreadyForwarded ← true
20:         end if
21:     end if
22: end procedure
```

The function sendRequest does the following. Let us assume that the requesting process is P_u. First, if checks if it is the root and there are no pending requests in its request queue. If this is indeed the case, then it grabs the lock and the process terminates there. Otherwise, P_u adds the request to its own resource queue. This is because this is a pending request that will get satisfied later. Then, the request is forwarded to the parent (one step towards the root). It is possible, as we shall see later, for a process to forward messages on behalf of other processes (in its subtree). If a message has already been sent (`alreadyForwarded` = **true**), then there is no need to send a message to get the token again.

When the parent node gets a ⟨`request`⟩ message, it calls the receiveRequest function. If the parent's request queue is empty and it has the token (is the

root), then this situation is quite straightforward. All that the parent needs to do is forward the token to the requester – process P_u in this case.

In the other case, if the parent is not the root (token holder), then it should forward the request in the direction of the root (to its parent). Of course, it is possible that the parent has already sent a request to obtain the token – on behalf of some other request. In this case, there is no need to send another request. Otherwise, it needs to send a new request in the direction of the root to obtain the token (refer to Line 18). This process continues till we reach the root. Note that in every step, we go one step closer to the root; hence, this process is guaranteed to terminate.

Let us now look at the next procedure, releaseToken. During this period, it is possible that several requests would have arrived at the root node and the requests would be queued in its request queue. The root node dequeues the first request and sends the token to it. Note that requests can only arrive from children of the root.

Algorithm 15 Raymond's tree algorithm: release the token

```
1:  procedure releaseToken(Process P_u)
2:      Forward the token to the process that is at the head of the request
    queue
3:      Dequeue the entry
4:      Update the parent pointer
5:  end procedure
6:  procedure receiveToken(P_v receives from P_u)
7:      P_v.alreadyForwarded ← false
8:      Dequeue the an entry from the request queue
9:      if P_v was the head of the request queue then
10:         Grab the lock and access the resource
11:         After using the resource, if the request is non-empty then return
    and wait
12:         Else, dequeue the request queue again.
13:     end if
14:     Forward the token to the dequeued entry. Update the parent pointer.
15:     if P_v's request queue is non-empty then
16:         send a fresh request to the current token holder
17:         P_v.alreadyForwarded ← true
18:     end if
19: end procedure
```

Algorithm 15 shows the code for releasing a token. Let us first take a look at the RELEASETOKEN function. When a process finishes executing the critical section (using the resource), it releases the token. The first step is to check the request queue. If there are no entries, then there is nothing to do. We need to just wait for a request to arrive. However, if there are entries, then there is a need to dequeue an entry and pass the token to it (see Line 3). This node now becomes the new root. It is the token holder and is also connected to the older root node via an edge. Assume that process P_u forwards the token to P_v. P_u was the old root and P_v is the new root. P_u updates its parent pointer. It now points to P_v.

When a process receives a token, it sets `alreadyForwarded` to *false* (the response to the token request has been received and there is no outstanding request), and then checks its request queue. If the request queue is empty, there is nothing to do. This trivial case had not been shown in function RECEIVETOKEN. Let us assume that the request queue was non-empty. P_v needs to dequeue the head of the queue. This entry could be P_v itself. In this case, P_v can access the resource (enter the critical section). Once it is done using the resource, it needs to dequeue its request queue again. If the queue is empty, then there is nothing to do. It can return from the RECEIVETOKEN function and just wait. Consider the other case, when there is an entry in the request queue of P_v. We dequeue an entry. P_v forwards the token to the dequeued entry and updates its parent pointer (same was done in the function RELEASETOKEN).

It is possible that there could be other requests in P_v's request queue. Given that it has forwarded the token, it needs to get it back. Thus, there is a need to issue a fresh request for the token in Line 16 for the token. We again set P_v.`alreadyForwarded` to *true*.

Proof of Correctness

The proof of mutual exclusion is trivial. There is only one token, and that can only be held by one process at any given point of time. Hence, it is not possible for two processes to be in the critical section at the same time.

Let us now look at the liveness properties. Ultimately a request from process P_w will become the head of all the request queues – when it becomes the oldest in the system. At that point of time, the token will need to come back to the process P_w, which had initially placed that request. This is because of the way we process received tokens. We always dequeue the request queue and act on the dequeued request first. This means that even if the request from P_w has been starving, the moment it becomes the oldest

in the system, the token will have to come to it.

Message Complexity

Let us consider a conservative worst case by looking at each phase of execution. The ⟨`request`⟩ message is sent over $O(log(n))$ links. For each request, a reply is sent by sending the token towards the original requester. This accounts for $O(log(n))$ more messages.

Sadly, more messages need to be sent because it is possible that there are other requests that have a higher priority. The token will be sent to them and a request needs to be sent to get the token back. Here, we need to carefully count the number of messages (see Line 16). Whenever, the token is sent in a certain direction, it is because there is a request in that direction that has the highest priority in the request queue of the token sender. Consider an example.

Process `P` needs the token. It sends a request to Process `Q`, which is the current token holder. `Q` sends the token back in the direction of `P`. However, a node in the middle running Process mR seeds that there is a request from another process `S` that is earlier. It thus sends the token in the direction of `S`. Given that it already has a request that was originally initiated by node `P`, it sends a fresh token request to get the token back (after `S` uses the token and possibly other nodes). Now, the question is how do we account for this message that Process `R` sends to get the token back? We can charge this message to the account of Process `S`. We can assume that whenever a token is sent in the direction of some process, then we also add the overhead of one request to get the token back. This simplifies things. In the path from `Q` back to `P`, there will be several such nodes that will send the returning token in other directions because they will have earlier requests from processes like `S`. However, in every such case the request that is made to get the token back is accounted for by a different process (not `P`).

We only add the messages that account for directly passing the token from `Q` to `P`, and 1 request per link (to get the token back), to Process `P`'s message complexity.

Hence, the total overhead is limited to $O(log(n))$.

Remark 22 *The Suzuki-Kasami algorithm requires $O(n)$ messages and the Raymond's tree algorithm requires $O(log(n))$ messages.*

4.3 Chubby Lock Service

Chapter 5

Distributed Algorithms

In the previous chapters, we studied several systems aspects of distributed systems including file systems, mutual exclusion, database systems, and the like. In this chapter, we move to study distributed algorithms. To this end, we will first outline the model used in designing distributed algorithms and the related perofrmance measures. Following this, we will show examples of distributed algorithms for a variety of problems such as leader election , graph traversals , and other problems from graphs such as coloring , dominating sets , routing , and the like.

5.1 The Model and Performance Measures

Recall that a distirbuted system can be seen as a collection of processors (nodes) with communication links across a subset of nodes. The links can be taken to be unidirectional or bidirectional. This allows us to model the distributed system as a graph G. The nodes of G are the processors of the distirbuted system and the edges of G are the communicaiton links of the distributed system. The topology of G corresponds exactly to the distributed system.

For a computation to be performed in such a distributed system, we note that nodes in a distributed system do not have access to shared memory, nodes have to rely on messages to perform the computation. Further, we often think of distributed sytsems as being asynchronous in nature. In the asynchronous case, it is not possible to specify an upper bound on the message delays. So, the algorithm has to also contend with unpredictable message deliveries as the computation happens.

On the other hand, it is easier to design distributed algorithms in a syn-

chronous distributed system. In this case, we can envision the computation as happening in synchronous rounds. In each round, each processor (node) can perform some local computation, send message to each of its neighbors, and recieve a message from each of its neighbors. This consitutes one logical round of computation of the distributed algorithm.

Currently, several distributed algorithms are known using the synchronous model. Fortunately, however, there exist constructors called as *synchronizers* that show how to convert distributed algorithms in the synchronous model to algorithms in the asynchronous model. In the rest of this chapter, unless otherwise mentioned, we deal with only synchronous distributed algorithms. We discuss the notion of synchronizers and how they allow one to transform an algorithm designed in the synchronous model to an asynchronous model in Section 5.9.

We now move to discuss performance measures for distributed algorithms. Consider a sequential algorithm such as merge sort [30]. There are several performance measures one can use to analyze the algorithm. Often talked about are the time complexity of the algorithm and the space complexity of the algorithm. Other measures that have been proved to be useful is the number of disk accesses made by the algorithm, and the like. Similarly, even for distributed algorithms, there exist several performance measures.

Some of the performance measures include the space complexity per node, and the system-wise space complexity. Another such measure is the time complexity per node. Here, time complexity is measured in terms of logical rounds needed by the algorithm. An equally imortant mesaure is the message complexity of the algorithm. In this case, one can consider the number of messages sent/received by a node and the size (volume) of the messages sent/received by a node. The message size affects the time taken to deliver the message in addition to the space needed to handle the message at the source and the destination nodes.

Based on the above discussion, we identify the following model to start designing distributed algorithms. Subsequnetly, we will also discuss other models that are used to design distributed algorithms. In the basic model we use, we consdier a synchronous setting so that computation can be seen as proceeding in logical rounds. As mentioned earlier, in each round, each node can perform some local computation, send a message to each of its (outgoing) neighbors, and receive a message from each of its (incoming) neighbors. We measure the performance of an algorithm in terms of the number of rounds needed, the size and number of messages, the local and global space needed by the algorithm.

To describe distributed algorithms, we use pseudcode that is similar in flavor to what is used in the rest of the book. We assume that each node is executing the pseudocode.

5.2 Algorithm for Graph Traversal

The first algorithm we study in the distributed setting is the graph breadth-first traversal, popularly known as BFS [30] . We consider that the links are bidirectional. There is also a designated node called the *source* and denoted s and all nodes know that s is the source of the traversal. At the end of the computation, we require that each node know its distance from s and also the parent on a path from the node to s.

The basic idea of the algorithm is to proceed in rounds. In each round, a set of nodes that are *marked* enrol their unvisited nodes and mark such nodes. During this process, such unvisited nodes are also assigned the *correct* level number that corresponds to the distance of such nodes from s. The parent is also set during this marking step. The computation stops when there are no further nodes to mark. If the graph is connected, then we see that all nodes will be marked. If the graph is disconnected, and there are nodes that are not reachable from s, the computation stops but nodes that are not reachable from s will remain unmarked.

Initially, the source node s sets its level number to be 0 and its parent to be s itself. The source then sends a MARK message to all its neighbors. The message will include the level number of the initiator as 0. The neighbors will mark themselves as nodes at Level 1 and propagate their MARK messages. Note that some node may now receive more than one MARK messages. Such a node picks any of the message sources as its parent and set its level number to be one more than the level number included in the MARK message.

Each node can terminate when it gets a valid level number and sends out the MARK message(s). This process repeats until all nodes are marked with a valid level number or there are no further nodes that are yet to terminate.

The pseudocode for this computation is in Algoritm 16. Figure 5.1 shows an example of the execution of the algorithm.

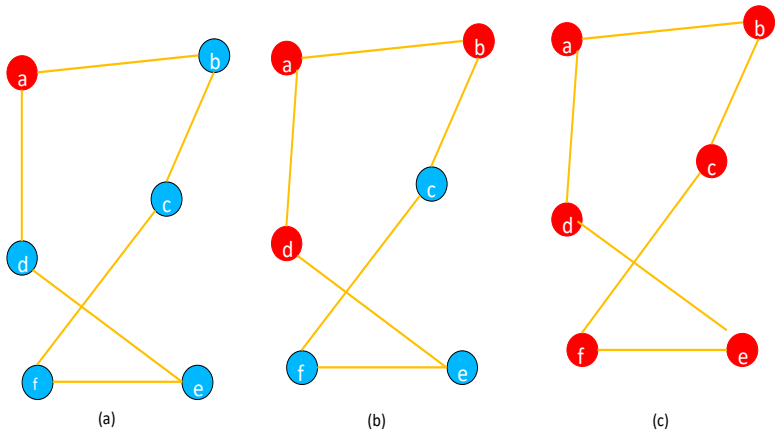


Figure 5.1: An example run of the synchronous BFS algorithm (Algoritm 16). Nodes in blue color are marked as visited and nodes in red color are unmarked. Node a is the source and hence nodes b and d are marked in the next round (see part (b) of the figure). Subseuqently, the other nodes are marked in the third round as shown in part (c) of the figure.

Algorithm 16 Synchronous BFS Algorithm

```
1:  procedure GRAPH-SYNCHRONOUS-BFS
2:      marked ← false, level ← 0; parent := -1;
3:      if u == source then
4:          marked ← 1; level ← 0;
5:          Send message ⟨Mark, u, 0⟩ to all neighbors of u;
6:      end if
7:      for round ← 1 to diameter do
8:          if marked == false then
9:              if ⟨Mark, v, levelv ⟩ is received then
10:                 parent ← v; marked ← 1; level ← levelv + 1;
11:                 Send message ⟨Mark, u, level ⟩ to neighbors;
12:             end if
13:         else
14:             Delete any Mark messages
15:         end if
16:     end for
17: end procedure
```

Analysis

Let us analyze Algorithm 16 in terms of its performance. As can be noticed, the loop in Line 6 runs for a number of iterations that equals the diameter of the graph[1]. This follows from the observation that the node farthest from the source is at a distance that is at most the diameter of the graph. (See also Question 5.1 to reduce the number of iterations possibly). Further, a node recieves (or sends) a number of mark messages that is up to the number of its neighbors. Therefore, the local and the global time complexity of the algorithm is in $O(\text{diameter} + \text{degree})$.

Since each node needs to know its neighbors, the local space complexity is in $O(\text{degree})$. Each Mark message contains the level number of the sender and the id of the sender. So, each message has a size in $O(\log n)$ when there are n nodes in the graph. Each edge carries at most two Mark messages.

In the simple algorithm presented, each node knows its parent via the assignment to the parent variable in Line 9. Question 5.1 asks you to modify Algorithm 16 so that each node knows its children nodes as well.

5.3 Maximal Independent Set (MIS)

One of the fundamental problems on graphs is to find independent sets. In a graph $G = (V, E)$,), an independent set is a set of nodes U, where $U \subseteq V$, such that for each i and j in U, $(i, j) \notin E$. Such an independent set U is called a *Maximal Independent Set* (MIS) if no proper superset of U is also an independent set. Figure 5.2 shows an example of a graph (Figure 5.2(a)) and two of its maximal independent sets (Figures 5.2(b)–(c)).

We inform the reader that the notion of an MIS should not be confused with the notion of a Maximum Indepdenet Set which asks for an independent set of the largest size in G. The problem of finding a maximum independent set is an NP-complete problem [46] whereas a maximal independent set can be found in $O(m + n)$ sequential time using a simple greedy algorithm.

Algorithm 17 shows a sequential greedy algorithm that can identify an MIS in a graph. This algorithm starts with an empty MIS and adds one vertex to the MIS under construction in each iteration of the while loop at Line 2. Notice that the size of the MIS depends on the choice of the vertex v in Line 3. See also the Questions 5.2–5.3 at the end of the chapter for more observations corresponding to Algorithm 17.

[1]Recall from graph theory that the diameter of a graph is the largest of the pairwise shortest distances across nodes in the graph

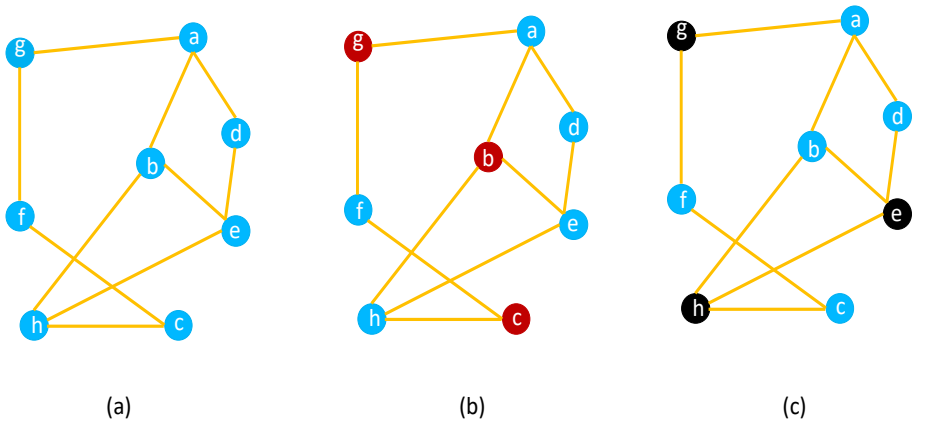


Figure 5.2: A graph in part (a) along with two of its maxiimal independent sets in parts (b) and (c) with nodes in the MIS colored red and black, respectively.

Algorithm 17 Sequential Greedy Algorithm for identifying an MIS.

1: $U \leftarrow \Phi$; $S \leftarrow V$;
2: **while** S is not empty **do**
3: Pick a vertex $v \in S$ and add v to U;
4: Delete v and all neighbors of v from S;
5: **end while**
6: Return U as the MIS;

Some more observations concering Algorithm 17 are in order. This algorithm continues to work in a distributed setting also. The message complexity of the algorithm is also quite small. We can pick v to be the smallest numbered vertex of S in Line 3. The membership of a vertex in S can be tracked via the use of appropriate flags. Question on the round complexity with the numbering scheme Show that there exists a numbering scheme that works best.

However, one drawback of Algorithm 17 is the number of iterations of the while-loop at Line 3. In the worst case, there can be $\Omega(n)$ iterations leading to an $O(n)$ number of rounds of the distributed version of the algorithm. We seek better distributed algorithms and one mechanism is to reduce the number of rounds needed to identify an MIS.

Determinisitc distributed algorithms for obatining an MIS have a rich history. From the seminal result of Awerbuch et al. [8] shows a deterministic

distributed algorithm requires $e^{\sqrt{\log n \log\log n}}$ rounds to obatin an MIS. The result of Awerbuch et al. [8] is subsequently improved over the years. The current best deterministic distributed algorithm for the MIS problem runs in $e^{\sqrt{\log n}}$ rounds [95]. All these results rely on network decomposition that partitions the graph into a small number of clusters of a small diameter. We refer the reader to [8,9,95] for a thorough treatment of network decomposition and its application to MIS. While this is smaller than the $O(n)$ round complexity of Algorithm 17 running in a distributed manner, one would seek better algorithms.

The fundamental limitation seems to be to identify a *large* set of vertices that all can be part of an independent set and making such a decision simultaneously. Since all nodes are alike, we view them as being *symmetric* and hence some mechanism is needed to break the symmetry amongst competing nodes to be part of an independent set. This problem is often termed *symmetry breaking* and is a fundamental problem in distributed computing.

While efficient deterministic algorithms are hard to come by, randomization has proved to be quite helpful in symmetry breaking algorithms. In this section, we study one of the classical algorithms for obtaining an MIS from Luby [84].

The basic idea of Luby's algorithm is to in every iteration find a suitably large set of nodes that can be added to the independent set. To do so, the algorithm uses randomization to identify a set R that is not necessarily independent but can be trimmed to get the set S. The challenge is to ensure that R is big enough and trimming R still leaves a big enough set S. Algorithm 18 shows the pseudocode.

To analyse Algorithm 18, we show that in each iteration of the `repeat` loop in Line 2, a constant fraction of the edges get deleted on expectation. Since the number of edges in no more than n^2, this implies an expected $O(\log n)$ rounds for the computation to finish.

In that direction, we make the following characterization of nodes and edges. A vertex v is *good* if at least 1/3rd of its neighbors have degree less than $d(v)$. A vertex v is bad if at least 2/3rd of its neighbors have degree at least $d(v)$. An edge e is good if at least one endpoint of e is good. Finally, an edge e is bad if both its endpoints are bad.

From the above characterization, we observe that good vertices have enough low degree neighbors. So, we hope that at least one such low degree neighbor is in S with good probability since such an event helps delete good vertices. This in turn helps delete good edges. Eventually, if we can show that there are enough good edges, then suffices if a fraction of them are deleted. The above is formalized in the following set of claims.

Algorithm 18 Randomized Algorithm to obtain an MIS.

```
1: I ← Φ;
2: repeat
3:     for all nodes v in parallel do
4:         if d(v) == 0 then
5:             add v to I and delete v from V;
6:         else
7:             mark v with probability 1/2d(v);
8:         end if
9:     end for
10:    for all edges (u, v) ∈ E in parallel do
11:        if both u and v are marked then
12:            unmark the vertex of lower degree;
13:        end if
14:    end for
15:    for all nodes v in parallel do
16:        if v is marked then
17:            add v to S
18:        end if
19:    end for
20:    I ← I ∪ S;
21:    Delete S ∪ N(S) from V, and all incident edges from E
22: until V is empty
```

Lemma 23 *For every good vertex v with $d(v) > 0$, the probability that some neighbor w of v gets marked is at least $1–e^{-1/6}$.*

Proof: Since v is good, at least $d(v)/3$ neighbors have degree at most $d(v)$. Let w be such a neighbor. Note that the probability that such a w gets marked is $\frac{1}{2d(w)}$ from Line 6 of Algorithm 18.

As $d(w) \leq d(w)$, $\Pr\{w \text{ gets marked}\} = \frac{1}{2d(w)} \geq \frac{1}{2d(v)}$.

Since nodes get marked independent of each other,

$$\Pr\{\text{no neighbor of } v \text{ is marked}\} = \left(1 - \frac{1}{2d(v)}\right)^{d(v)/3} = e^{-1/6}.$$

□

Lemma 24 *If a vertex w is marked, then $\Pr\{w \in S\} \geq 1/2$.*

Proof: If w is marked, then $w \notin S$ only if some high degree neighbor of w is also marked. In such a case, w unmarks itself in Line 11 of Algorithm 18.

Each such high degree neighbors of w is marked with probability at most $1/2 \cdot d(w)$. The number of such high-degree neighbors is at most $d(w)$.

Hence,

$$\begin{aligned}
\Pr\{w \in S\} &= 1 - \Pr\{w \notin S\} \\
&= 1 - \Pr\{\exists u \in N(w), d(u) \geq d(w), u \text{ marked}\} \\
&= 1 - 1 - |u \in N(w), d(u) \geq d(w), u \text{ marked}|1/2 \cdot d(w) \\
&\geq 1 - |u \in N(w)|1/2 \cdot d(w) = 1–d(w) \times 1/2 \cdot d(w) \\
&= 1/2.
\end{aligned}$$

□

The above two lemmata directly give the following observation.

Lemma 25 *Let v be a good vertex. Then, $\Pr\{v \text{ is deleted}\} \geq (1 - e^{-1/6})/2$.*

Proof: Combine Lemmata 23 and 24. □

Finally, we bound the number of good edges in the following lemma.

Lemma 26 *At least half the edges are good.*

Proof:

For every bad edge e, associate a pair of edges via a function $f : E_B \to \binom{E}{2}$ such that for any two distinct bad edges e_1 and e_2, $f(e_1) \cap f(e_2) = \Phi$. Such a mapping will allow us to conclude the proof since only $|E|/2$ such pairs exist.

We define the function f as follows. For each edge $(u, v) \in E$, orient it towards the vertex of higher degree. Consider a bad edge $e = (u, v)$ oriented towards v. Since v is bad, the out-degree of v is at least twice its in-degree. So, there exists a way to pick a pair of edges for every bad edge. □

Using all the above lemmata, the following theorem concludes our proof.

Theorem 27 *In each iteration, a constant fraction of edges are deleted on expectation.*

Proof: Note that at least half the edges are good accoridng to Lemma 26. Further, from Lemma 25, a good edge is deleted with probability at least $(1 - e^{-1/6})/2 \geq 0.07$. □

We can extend Theorem 27 to further show that Algorithm 18 runs in $O(\log n)$ rounds with high probability[2]. Notice that the messages sent in any step of Algorithm 18 have a size in $O(\log n)$.

The analysis presented here is typical of several distributed algorithms. In general, the goal is show that enough progress happens over a certain number of rounds. With each round of the algorithm being independent of the other in terms of the probabilistic choices that nodes make, such a condition on progress helps. Another feature of the analysis is to relate the progress of a node as a consequence of the actions of its neighbors.

There has been a lot of progress on the MIS problem in the distributed setting since the classical algorithm of Luby [84]. The notes at the end of this chapter traces some of these developments.

5.4 Other Graph Algorithms

While MIS and its associated problems such as ruling sets [14, 19, 74] are very popular in the distributed algorithm research community, there are other graph problems of huge interest. Some of these include the following.

5.4.1 Vertex Coloring

The problem of vertex coloring is to assign colors (labels) to vertices such that no two neighbors receive the same color. Such a coloring is called as a proper coloring. The minimum number of colors needed to achieve such a coloring is the *chormatic number* of the graph G, denoted $\chi(G)$. It is well-known that finding $\chi(G)$, or approximating $\chi(G)$ up to $O(n^\epsilon)$, is NP-hard.

[2]The term with high probability refers to a probability that is in $1 - O(1/n^c)$ for some constant c

On the other hand, there exists a very simple sequential algorithm to obtain a proper coloring using $\Delta + 1$ colors where Δ is the degree of the graph.

Given that the problem of obtaining a proper $\Delta + 1$ coloring is closely related to that of the MIS problem, the lower bound for deterministic algorithms concering MIS [8] applies to the coloring problem also. Hence, distributed algorithms to obtain a proper $\Delta + 1$ coloring however need randomization.

One of the earliest randomized algorithms for obtaining a $\Delta + 1$ coloring is through simple modifications to the MIS algorithm of Luby. Assuming a palette of 2Δ colors, each node that is yet uncolored picks an available color from its palette independently and uniformly at random. If the choice of a node u conflicts with the choice of a neighbor v, then both u and v relinquish their choice and stay uncolored. Nodes that make a choice that is not in conflict with the choice of their neighbors get colored with their choice and inform their uncolored neighbors. These neighbors will delete that color from their palettes. This process repeats until all nodes obtain a color.

A simple analysis shows that the above process requires $O(\log n)$ rounds with high probability. (See Questions aa and bb). Barenboim et al. [14] present further progress on this problem wherein the algorithm runs for $O(\log \Delta + \sqrt{\log n})$ rounds.

5.4.2 Graph Spanners

Let $G = (V, E)$ be a weighted graph. For any pair of vertices $u, v \in V(G)$, let $d_G(u, v)$ denote the shortes distance from u to v in G. A t-spanner of G is a subgraph $H = (V, E_H)$ of G such that $d_H(u, v) \leq t \cdot d_G(u, v)$. In other words, distances in H are no more than t times the distances in G. The parameter t is called as the stretch of the spanner.

The interesting case is when E_H is much smaller in size compared to that of $E(G)$. Spanners find applications in settings such as communication networks and routing tables where one is interested in bounds on the pairwise distances in a graph and not the exact distances.

One typical way of building a spanner is as follows. For every edge (u, v) in G, we want to find a path in H such that $d_H(u, v) \leq t \cdot d_G(u, v)$. We maintain the property that for every edge $(u, v) \notin H$, vertices u and v are connected in H by a path $P_H(u, v)$ of at most t edges, and the weight of each edge on $P_H(u, v)$ is not more than that of $w_G(u, v)$. This property can be used to show the following property of H. If P is a path in G consisting of edges $(u_0, u_1), (u_1, u_2), \cdots, (u_{k-1}, u_k)$, then $wt_H(P) = \sum_i wt_H \cup_i (u_i, u_{i+1}) \leq$

$\sum_i t \cdot wt_G(u,v) \leq t \cdot wt_G(u,v)$.

Since H is a subgraph of G, we cannot add new edges. We have to only see how to delete certain edges of G in creating H. In the following, we will study a speical case of an algorithm due to Baswana and Sen [15] that shows how to build a 3-spanner. The algorithm naturally extends to other (odd) values of t.

The Algorithm of Baswana and Sen

The algorithm of Baswana and Sen produces a 3-spanner H of a grpah G such that H has no more than $n^{3/2}$ edges. Based on the earlier observations, the algorithm has to identify edges of G that will *not* be included in H. Since H has $n^{3/2}$ edges, for any vertex $v \in G$ with a degree up to $\sqrt{n}$, all its edges can be added to H. So, in the following, we consider only vertices that have a degree beyond $\sqrt{n}$ in G.

The algorithm proceeds by making appropriate clusters and uses the clusters to know which edges to keep and which edges to discard. To start with, let $E' := E$ and $E(H)$ be empty. Edges are removed from E' and some of the edges in E' are added to $E(H)$. The algorithm runs over two phases described below.

In Phase I, the clusters are formed. To this end, we choose a set R of cluster centers by adding each node v in G to R with a probability of $1/\sqrt{n}$ independently. Each $v \in R$ is called as the center of the cluster. We do the following with every vertex $w \notin R$.

- If v is not adjacent to any vertex in R, we move every edge incident on w to $E(H)$.
- If w is adjacent to one or more sampled vertices, let x in R be such that (w,x) has the smallest weight among incident edges at w with the other end point in R. Move the edge (w,x) to $E(H)$ along with all edges incident at w and have a weight less than that of (w,x). Add the vertex w to the cluster with x as the center. Finally, discard all those edges (u,v) from E' where u and v are not sampled and belong to the same cluster.

After Phase I, consider the edges in E' that are now left over. Let V' be the vertices that are end points of edges in E'. Each vertex in V' is either a sampled vertex or adjacent to some sampled vertex. Further, V' is partitioned into disjoint clusters each centered around some sampled vertex.

In Phase II, the algorithm joins clusters with other clusters as follows. Let v be in V'. Let $E'(v,c)$ be the edges from the set E' incident on v from a cluster c. For each cluster c incident to v, we move the least-weight edge from $E'(v,c)$ to $E(H)$. We discard the remaining edges.

It is easy to see that the above steps can be implemented in a distributed fashion where each node performs the actions as described. See also Question 5.5. In the rest of the discussion on the algorithm, we show how to analyze the algorithm with respect to the number of edges in H and the stretch of the obtained spanner.

Lemma 28 *The expected number of edges added to H in Phase I of the algorithm is $O(n^{1.5})$.*

Proof: We argue that each node adds no more than $O(\sqrt{n})$ edges to H in Phase I. Let us first consider nodes that are not sampled and also not a neighbor of any sampled node. Let $v$be such a node. If v has more than $\sqrt{n}$ neighbors, then the probability that none of these $\sqrt{n}$ neighbors is sampled is at most $(1-1/\sqrt{n})^{\sqrt{n}} \leq 1-1/e$. Thus, if v has a degree at least $\sqrt{n}$, then v is a neighbor of some sampled node. It therefore holds that the expected number of edges that a node that is not sampled and is not a neighbor of any sampled node adds to H in Phase I is bounded by $O(\sqrt{n})$. The total number of such edges added to H is Phase I is hence at most $O(n^{1.5})$.

Let us now consider nodes v that are a neighbor of some sampled node w. In this case, we consider the situation that v is not sampled but w is sampled. We argue that v has on expectation v has $O(\sqrt{n})$ neighbors that are not sampled and have a weight less than the weight of (v,w). To do so, let us imagine the neighbors of v ordered by weight. In such an ordering, according to the algorithm, w is the *first* neighbor of v that is sampled. So, the position of w cannot be beyond $O(\sqrt{n})$ since that means that none of the prior neighbors of v are sampled. Thus, each such v also adds $O(\sqrt{n})$ edges to H. We conclude that the expected number of edges added to H in Phase I is in $O(\sqrt{n})$.

Let us move to Phase II of the algorithm. In this phase, we add one edge per cluster center, node pair. Since the expected number of cluster centers is $\sqrt{n}$, we can limit the number of edges added in Phase II to $O(\sqrt{n})$.

The lemma follows. □

We now move to show the bound on the stretch of the construction.

Lemma 29 *The graph H is a 3-spanner of G.*

Proof: We will show that for every edge (u,v) not in $E(H)$, u and v are

connected by a path of at most three edges in H, and the weight of each edge in this path is no more than that of the weight of (u, v).

We start by observing that if an edge $(u, v) \in E(G)$ is not present in $E(H)$ at the end of the first phase, then the weight of edge (u, v) is greater than or equal to the weight of the edge between v and w, where w is the center of the cluster to which v belongs.

There are two cases to consider. Let u and v belong to the same cluster. From the above observation, it holds that there is a path $u--x--v$ in the cluster with the weight of (u, x) and the weight of (x, v) at most the weight of (u, v). So, the claim holds.

In the other case, we let u and v belong to two different clusters. Clearly the edge (u, v) was removed from E' during Phase II. Suppose it was removed while processing the vertex u.

Let v belong to the cluster centered at $x \in R$. In the beginning of Phase II, let $(u, v') \in E'$ be the least weight edge among all the edges incident on u from the vertices of the cluster centered at x. So, the edge (u, v') gets added to $E(H)$ during the processing of vertex u in the second phase of our algorithm.

The edge (u, v') in $E(H)$ creates the path $u--v'--x--v$ in $E(H)$. Each of these edges has a weight at most that of (u, v). So, the weight of this path is at most three times the weight of (u, v).
□

The algorithm of Baswana and Sen that we described above extends to obtain a $(2t + 1)$-spanner also for any positive integer $t \geq 1$. The idea is to repeat the process of forming the clusters (Phase I) for t iterations. This is followed by the approach of Phase II where we join the clusters. The algorithm obtains a $(2t + 1)$-spanner of size $O(tn^{1+1/t})$ in expected time $O(mt)$. Further, the simple and generic approach of the algorithm of Baswana and Sen allows the algorithm to work in multiple computational models in addition to the distirbuted computing model such as the PRAM model and the external memory model.

5.4.3 Facility Location

5.4.4 Minimum Dominating Sets

Given a graph $G = (V, E)$, a dominating set U is a subset of V such that every node $u \in V$ is either in U or is a neighbor of some node in U. The Minimum Dominating Set (MDS) asks for finding such a dominating set U of the smallest size. Figure 5.3 shows an example of a graph and its MDS.

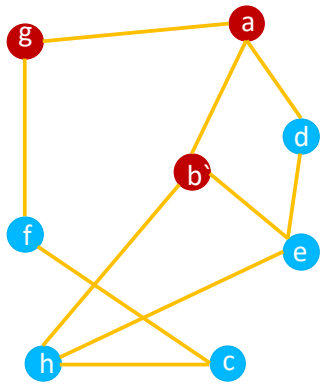


Figure 5.3: A graph with nodes colored in red part of an MDS.

The MDS problem is an NP-complete problem [46] and is closely related to the classical set cover problem.

For both the above problems, there exists a simple greedy sequential algorithm that works as follows in the case of MDS. In each iteration, starting with an empty set S, add one node u to S where u covers the most number of yet uncovered nodes. In the case of set cover, in each round we add a set that covers the most number of yet uncovered elements. This greedy algorithm achieves an approxmiation ratio of $O(H_\Delta)$ where Δ is the maximum degree of a node in G. In the case of set cover, Δ refers to the maximum number of elements in a set. We refer the reader to [67] for more details on the greedy seuqential algorithm and the approximation ratio achieved by a greedy sequential algorithm.

As can be noted, the sequential algorithm adds one element in each iteration to the output under construction. Since the number of nodes in a minimum dominating set can be in $\Theta(n)$ where n is the number of nodes in the graph, the sequential algorithm runs for $\Omega(n)$ iterations. To design an efficient distributed algorithm, we need to identify multiple nodes that can be added to the dominating set under construction so as to reduce the number of iterations needed.

One possibility is to mimic the greedy seqeuntial algorithm as follows. We let each node u compute its *span*, which is the number of nodes that u covers. Each node u communicates its span to all nodes that are at a distance of at most two from u. A node u joins the dominating set if its span is a local maximum in its 2-neighborhood. To break ties, we can use the tuple of span and the node id. Unfortunately, this simple translation fails to produce an efficient distributed algorithm even though the algorithm

does build a dominating set. The algorithm requires $\Omega(\sqrt{n})$ rounds in some graphs, notably the caterpillar graph [116]. Simple ways to remedy the situation by using a rounding of the span values to the nearest power of 2 also fail to work on some classes of graphs [66].

Jia et al. [66] present a refined algorithm that works as a greedy algorithm that we describe in the following. The algorithm uses randomization. Algorithm 19 shows the pseudocode.

Algorithm 19 Randomized Algorithm to obtain an approximate MDS.

1: /*Calculate the span*/
2: Each node u that is yet uncovered calculates span(u), which is the number of uncovered nodes that u covers. The value span(u) is rounded to the nearest power of 2.
3: /*Select Candidates*/
4: A node u is marked as a candidate if span(u) is at least span(w) for all w that are within a distance of at most 2 from u. For each candidate u, cover(u) refers to the set of nodes that u covers.
5: /*Calculate Support*/
6: For each node u that is yet uncovered, compute support(u) as the number of candidate nodes that cover u.
7: /*Selection Step*/
8: For each node u that is a candidate node, mark v as a dominator node with probability 1/median(v), where median(v) is the median of the support of nodes in cover(u).

Jia et al. [66] show that Algorithm 19 runs in $O(\log \Delta)$ rounds in expectation and produces a dominating set with an approximation ratio of $O(H_\Delta)$. Key insights from the analysis and the algorithm design are summarized in the following.

The size of the dominating set that the algorithm produces depends on how many nodes that cover a given node u are in the dominating set in general. This number should be at least 1 and ideally should not be too many. To satisfy this condition, one can imagine that each node in support(u) is added to the dominating set with a probability of 1/support(u). While this ensures that u is covered with a constant probability by at least one node, ensuring that this property holds across all nodes simultaneously is difficult.

However, the choice of the probability in the Selection Step intuitively works for the following reason. The probability of adding a candidate u to the dominating set is set to be the reciprocal of the median support of all the

nodes that cover u. Thus, if a node u has a support bigger than the median support among all nodes that cover u for more than half the nodes that cover u, then we can ensure that u is covered with a constant probability. This works to bound the expected number of rounds. For analysing the approximation ratio achieved, Jia et al. argue that the expected ratio of the number of nodes covered to the sum of the spans of the nodes in the dominating set is constant.

For bounding the number of rounds needed, Jia et al. [66] use an approach that is similar in spirit to that of the analysis of the MIS algorithm from Section 5.3. The argument establishes that in each round there exist enough *good* nodes and each such good node will be covered with a constant probability in each round. The characterization is done as follows. For each candidate v, consider cover(v) in sorted order according to the support of nodes in cover(v). The sorted entries in cover(v) can be visualized as having a top half, denoted $T(v)$, and a bottom half, denoted $B(v)$. For a given candidate v a node u is *favorable* for v if $u \in T(v)$. A node u is *good* if u if at least $\frac{\text{support}(u)}{4}$ candidate nodes that cover u are favorable for u. The intuition here is that if a node is good, then with a constant probability, such a node will end up being covered in a round. See Questions 5.6 and 5.7 for a hint to show that the above intuition indeed works.

5.5 Design Methodology for Distributed Algorithms

From the limited examples discussed in the preceeding sections, it appears that each distributed algorithm has to be designed bottom-up with very little algorithmic tools in hand. On the other hand, one observes that designing sequential algori thms is often undertaken as an exercise where a set of existing design paradigms are brought into play to ease the process of solution design and analysis. Fortunately, such is the case with distributed algorithms also. In this section, we outline some of the tools for designing and analyzing distributed algorithms.

5.5.1 Tree Traversal Based Computations

In this mechanism, the computation is visualized as happening along the edges of a full binary tree of the nodes. The tree is only a virtual design and analysis aid and is almost always never constructed. The computation can proceed in two ways:

- Upcast: In this mechanism, each node of the tree holds a value and

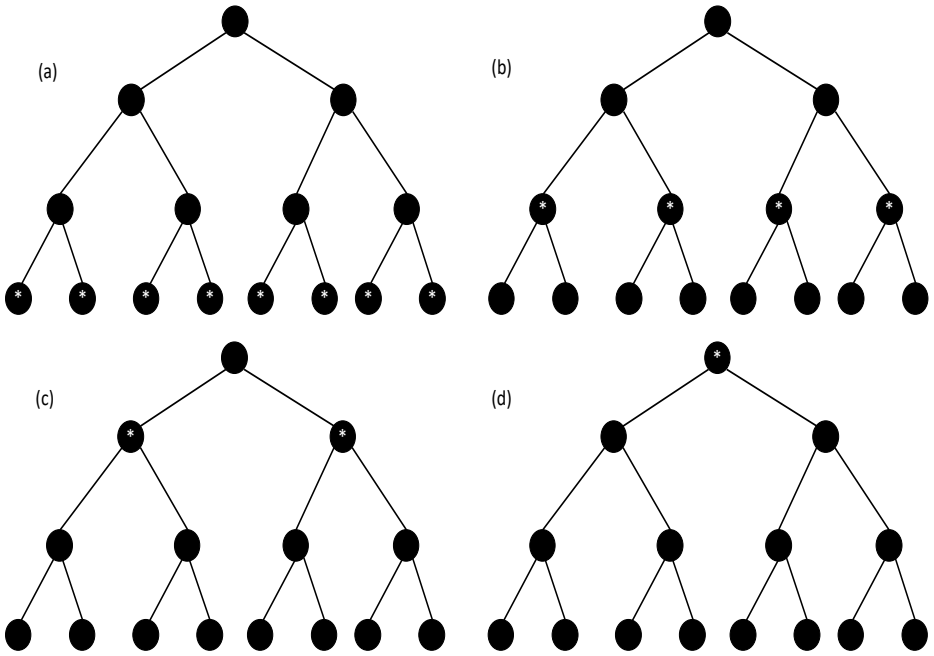


Figure 5.4: Upcast Example. A * mark on the node corresponds to nodes that send messages in that round.

applies an operator based on the values at the children of the node. Subsequently, the result is passed to the parent node in the tree and the node halts (terminates) its computation. The computation reaches the root of the tree and stops.

Figure 5.4 shows an example of how this mechanism works. Any node send one message to its parent, marked with a * on the node in the figure, once it receives inputs from its children.

- Downcast: In this mechanism, seen as the opposite of Upcast, the root node holds a value and passes down the value, or a function of the value, to its children. Each of these children in turn pass the value down to their children in the next round and terminate. Computation finishes when the values reach the leaf nodes in the tree.

In both of these mechanisms, it is not necessary that a parent and its children are neighbors in the graph corresponding to the underlying distributed system. Each of these mechanisms have a message complexity of $O(n-1)$ since there are $n-1$ edges in the tree. The number of rounds needed

depends on the diameter of the underlying graph since each message may need up to O(diameter) rounds to reach from the source to the destination. However, there are only $O(\log n)$ sequence REWORD of messages since the depth of the tree is in $O(\log n)$.

In some cases, it is not necessary that the tree is a full binary tree. In this case, there will be $O(h)$ phases of messages sent in each of the two mechanisms where h is the height of the tree.

Applications of Upcast and Downcast

Downcast is also popularly known as *broadcast* and is useful in settings where information has to be diffused from a source node.

5.6 Models of Distributed Computing for Algorithm Design

Distributed algorithms is an active and popular area of research. For a formal design and analysis of distributed algorithms, the model on which the algorithm works has to be clearly specified. There exist several models and in this section, we review some of the popular models.

5.6.1 The LOCAL model

In the LOCAL model, nodes can send and recieve messages of arbitrary size in each round. In other words, there is no limit on the message size. The intention of algorithm design in the LOCAL model is to see the number of rounds required ny the distributed algorithm to perform the computation as the input being distirbuted across nodes. The cost of local computation is also ignored.

A naive algorithm to solve *any* (solvable) problem in this model is therefore to gather the graph at a single node, say node with number 1. If the entire graph is at a single node, then that node can perform the required local computation and send the answer to all the other nodes. Since each of these two steps can be accomplished in a number of rounds equal to the diameter of the graph, the entire algorithm runs in O(diameter) rounds. Of interest is therefore algorithms that better this trivial solution. Indeed, for several problems on graphs such as connected dominating sets, vertex coloring, edge coloring, there are efficient (poly)logarithmic round distributed algorithms in the LOCAL model. On the other hand, there are problems such as the minimum spanning tree for which Ω(Diameter) is a lower bound

on the number of rounds needed. In fact, the best known lower bound for the MST problem with deterministic algorithms is $O(\text{diameter} + \sqrt{n/\log^2 n})$ due to the results of

The problem of MIS in the LOCAL model has attracted wide research attention since the result of Luby [84]. Kuhn et al. [75] show that *any* distributed algorithm for identifying an MIS in the LOCAL model needs at least $\Omega(\log n/\log\log n)$ rounds. This lower bound result indicates that the algorithm of Luby [84] is close to the best possible round complexity.

5.6.2 The CONGEST Model

The CONGEST model is another popular model for designing distributed algorithms. This name appears in the book by Peleg (Peleg, 2000 book on Distributed computing). This model shares several commonalities with the LOCAL model such as the graph resembling the distributed system, being fault free, synchronous, and the like. However, in this model, there is a upper limit on the message size on any link. In particular, the message size is usually limited to $O(\log n)$ bits where n is the number of nodes. This size is enough to convey the number of a node or some such polylogarithmic information about a node or edge of the graph.

The fundamental question that algorithms in this model have to answer is to see how to solve problems with the limited communication capacity of each link in each round. The limit on the link bandwidth essentially means that unlike in the LOCAL model, it is not possible to gather the entire graph at a single node in $O(Diameter)$ rounds. Therefore, the naive approach of gathering the graph and then solving locally does not work in the CONGEST model. In fact, it is known that there are problems on graphs that do not have any $o(n)$ round algorithm in the CONGEST model. (Question **??** asks you to find such a problem.)

On the other hand, for problems such as the All-pairs-shortest-paths, Holzer and Wattenhofer present an optimal algorithm that runs in $O(n)$ rounds in the CONGEST model. [**?**].

5.6.3 The k-Machine Model

We now describe the adopted model of distributed computation, the *k-machine model* (a.k.a. the *Big Data model*), introduced in [71] and further investigated in [13, 26, 39, 63, 72, 96, 97, 100]. The model consists of a set of $k \geq 2$ machines $\{M_1, M_2, \ldots, M_k\}$ that are pairwise interconnected by bidirectional point-to-point communication links. Each machine executes

an instance of a distributed algorithm. The computation advances in synchronous rounds where, in each round, machines can exchange messages over their communication links and perform some local computation. Each link is assumed to have a bandwidth of B bits per round, i.e., B bits can be transmitted over each link in each round; unless otherwise stated, we assume $B = \Theta(\text{polylog}N)$ (where N is the input size), although our bounds can be easily rewritten in terms of B [96][3]. Each machine has its own (internal) memory (assumed to be unlimited[4]) which it can access for free (any number of its own memory locations can be accessed simultaneously in one round). Machines do not. share any memory and have no other means of communication.

We assume that each machine has access to a private source of true random bits. We say that algorithm $\mathcal{A}$ has ϵ-*error* if, in any run of $\mathcal{A}$, the output of the machines corresponds to a correct solution with probability at least $1 - \epsilon$. To quantify the performance of a randomized (Monte Carlo) algorithm cA, we define the *round complexity of* cA to be the worst-case number of rounds required by any machine when executing $\mathcal{A}$.

In each round, each machine some (local) computation in parallel which depends on its current state and the messages that it received in the previous round; it can then "send" messages to other machine (that will be received at the next round), modify the state of its vertex and its incident edges. Messages are typically sent along outgoing edges, but a message may be sent to any vertex whose identifier is known (note that this is easy to accomplish since the identifier tells which machine a particular vertex is hashed to — cf. Section **??**). We note that the computation and communication associated with a vertex is actually performed by the *machine* that is responsible for processing the vertex (though it is easier to design algorithms by thinking that the vertices are the ones performing computation [1, 86]). Local computation within a machine is ignored since the focus is on the exchange of messages between machines, which is the costly operation. The *round complexity* of an algorithm is the maximum number of rounds until termination.

[3]There is an alternative (but equivalent) way to view this communication restriction: instead of putting a bandwidth restriction on the links (which increases with the number of machines), we can put a restriction on the amount of information that each *machine* can communicate (i.e., send/receive) in each round [96]. This is similar to the restriction imposed in the popular Map-Reduce/MPC models as well [**?**, 69]. In such model, the memory and bandwidth of each machine are restricted to sublinear in the size of the input; this is equivalent to choosing k sublinear in the input size in the k-machine model.

[4]In many k-machine algorithms, the memory usage is limited, essentially proportional to the size of the input allocated to that machine.

5.6.4 The Massively Parallel Computing (MPC) Model

The MPC model is defined by a set of machines, each having at most S words of memory. The machines are connected to each other via an all-to-all communication network. Communication and computation in this model are synchronous. In each round, each machine can receive a total of up to S words from other machines, performs local computation, and sends up to S words to other machines. The key characteristic of the MPC model is that both the memory upper bound S and the number of machines used are assumed to be strongly sublinear in the input size N, i.e., bounded by $O(N^{1-\epsilon})$ for some constant ϵ, $0 < \epsilon < 1$. This characteristic models the fact that in modern large-scale computational problems the input is too large to fit in a single machine and is much larger than the number of available machines.

For graph problems, the input size is $\tilde{O}(m+n)$ where m is the number of edges and n is the number of nodes of the input graph. Thus, $O((m+n)^{1-\epsilon})$, for some constant ϵ, $0 < \epsilon < 1$, is an upper bound on both the number of machines that can be used and the size S of memory per machine. It turns out that the difficulty of graph problems varies significantly based on how S relates to the number of nodes (n) of the input graph. Specifically, three settings for S have been considered in the literature.

- Strongly superlinear memory: In this regime, we set $S = O(n^{1+\epsilon})$. With such large space at each machine, it is important to note that for hte model to be realistic, we need the graph to be highly dense, i.e., $m \gg S \gg n$ such that S is strongly sublinear in m. Even as the input graph is dense, the fact that each machine has $O(n^{1+\epsilon})$ local memory makes this model quite powerful. For example, in this model, problems such as minimum spanning tree, MIS, and 2-approximate minimum vertex cover, all have $O(1)$-round algorithms [59, 70].

- Near-linear memory: In this setting, each machine has space just enough to store $O(n)$ infomration and hence we set $S = \tilde{O}(n)$. With this limited space, solutions become harder in this regime. However, recent progress in this regime has shown that symmetry breaking problems such as MIS, approximate minimum vertex cover, and maximal matching can be solved in $O(\log \log n)$ rounds [25, 32, 47, 48]. In addition, Assadi et al. [7] presented an $O(1)$-round algorithm for $(\Delta + 1)$-vertex coloring.

- Strongly sublinear memory: In this setting, we consider the situation where each machine has very limited space characterized as $S = O(n^{\epsilon})$.

Due to this limitation on space, solutions get much harder in this setting. The question of designing sublogarithmic-round algorithms for certain graph problems in this regime is an important research direction. For example, it is conjectured that the problem of distinguishing if the input graph is a single cycle vs two disjoint cycles of length $n/2$ requires $\Omega(\log n)$ rounds [49,119]. However, even in this regime, Ghaffari and Uitto [50] and Kothapalli et al. [73] recently showed that an MIS and a ruling set does have a sublogarithmic-round algorithm, running in $\tilde{O}(\sqrt{\log \Delta})$ and $\tilde{O}(\log^{1/6} \Delta)$ rounds, where Δ is the maximum degree of the input graph, respectively.

5.7 Leader Election

In distributed algorithms, there is often a need to elect a leader – some kind of coordinating process. Skeptics may quickly argue that electing a leader is against the philosophy of distributed systems. After all, we want all the processes to be symmetric, and do the same things and have the same responsibilities. Furthermore, if the leader goes down because of a fault, we will be left leaderless and it may be very hard to find if the leader is just slow or is genuinely out of service (similar to the classic FLP result, see Section 6.2). This is undoubtedly a valid criticism. However, there are algorithms and that too very popular ones that rely on the existence of leaders fully being aware that leaders may suffer from faults and also go out of service during the execution of the protocol. At that point of time a new leader may be elected, but it is very much possible that the earlier leader suddenly wakes up and for a short period there are two leaders in the system. Keeping these disclaimers in mind, having a leader is often a good option and it simplifies the protocol significantly.

5.7.1 Chang-Roberts Algorithm

Algorithm 20 Chang-Roberts Algorithm

```
1: procedure runInitiator
2:     if state = lost then return ▷ Let someone else become the leader
3:     end if
4:     state ← find
5:     send ⟨u⟩ to next(P_u)                          ▷ Send u along the ring
6:     while state ≠ leader do
7:         receive(P_v);
8:         if u = v then
9:             state ← leader
10:        else if v ¡ u then
11:            if s tate = find then state ← lost
12:            end if
13:            send v to next(P_u)
14:        end if
15:    end while
16: end procedure
17: procedure runOtherProcesses
18:    while true do
19:        receive request ⟨r⟩ from P_v
20:        send ⟨r⟩ to next(P_u)
21:        if state = asleep then
22:            state ← lost              ▷ Lose interest in becoming a leader
23:        end if
24:    end while
25: end procedure
```

Let us describe the simplest algorithm in this space, the Chang-Roberts algorithm. The processes are arranged in a ring (like a DHT); however, unlike a DHT messages can only be sent to immediate neighbors. All the processes are numbered from 1 to n. Note that they may not be contiguously numbered along the ring. Moreover, several processes may simultaneously initiate the process of leader election. Out of them, let the convention be that the process with the *smallest id* that is interested to be a leader shall win the election. This is an arbitrary choice, however, it makes our life simple. Algorithm 20 describes the protocol from the point of view of a given process that initiates leader election for itself. Let us refer to this

process as $\mathtt{P_u}$.

For the rest of this section, we shall assume that the current process that initiates the protocol or executes a method is referred to as process $\mathtt{P_u}$, whereas the *other* process that is sending the message is referred to as $\mathtt{P_v}$. This is the standard convention that we shall use.

Now, consider the method RUNINITIATOR. If the state is lost, it basically means that most likely some other leader has been elected. Given that an elected leader cannot be displaced and there cannot be two leaders at a time, the initiator has no choice but to stop the process of leader election. In this case, it knows that another leader is definitely either elected or going to be elected in some time. The system will not remain leaderless.

If this is not the case, we set the state to find. This means that we are in the process of finding the leader. The initiator starts the algorithm be sending its id $\langle \mathtt{u} \rangle$ to its neighbor, $\mathtt{next}(\mathtt{P_u})$.

As long as the state is not set to leader, the algorithm continues. A process continuously listens for messages. Note that in the ring topology a process can only receive a message from its immediate neighbor and there is only one such neighbor. Assume it receives the message $\langle \mathtt{v} \rangle$. There are several cases here.

If $\mathtt{u} = \mathtt{v}$, then it means that the message initiated by process $\mathtt{P_u}$ went through the entire ring and then was received back. This means that no process on the ring absorbed the message. We can thus assume that the message has the highest priority and there is no other claimant that has a higher priority. In this case, the state can be set toleader.

Next, assume that $\mathtt{v} < \mathtt{u}$. Let us then analyze the state. If the state is equal to find, then it means that the process of searching for a leader has not terminated. Given that we want to elect that process as the leader, which has the smallest id, we can conclude that $\mathtt{P_u}$ has lost the leader election when it is found that the request from $\mathtt{P_v}$ has a higher priority. This is indeed the case when mv¡u. We can set the state to lost. Henceforth, if any message is received from the neighboring node, it needs to be dutifully forwarded along the ring (Line 13).

Note that if none of these conditions hold, which basically means that $\mathtt{v} > \mathtt{u}$, we can absorb its message and remove it from the ring. Given that we know that there are other high priority requests in circulation, there is no need to forward such requests. Either the request from $\mathtt{P_u}$ is alive because $\mathtt{P_u}$ has not received any message with a higher priority (lower id) or the state has been set to lost. In the latter case when the state has been set to lost, it means that there is a request with higher priority than the request from $\mathtt{P_u}$. In either case, it means that there is a request in the ring whose priority

is more than the request from P_v and thus the request from P_v needs to be removed from the ring.

Let us now consider what other processes do, which are not initiators, i.e., claimants to become leaders (see the method runOtherProcesses). This method is very straightforward. Assume a request $\langle r \rangle$ is received from P_v. We just forward it to $next(P_u)$.

If the state of the node is asleep, we set the state to lost. This is because the process has forwarded a message at this point, and we do not want it to be out of the leader election race.

Proof

Let us assume that to the contrary, two processes have been elected as leaders. Let them be processes P_u and P_v. Let $u < v$. This means that process P_u has a higher priority.

Consider the case when P_v's request arrived at P_u (Line 10). At that point of time, if P_u was still asleep, then its state would be set to lost (as per Line 22). This means that in the future, it will not show any interest in becoming a leader. This is because it has been superseded by a request that was issued before (either P_v's request or something earlier than that). Given that it will not subsequently initiate any leader election (see Line 2), it cannot be elected as a leader. Hence, it is not possible that P_u and P_v are simultaneously elected as leaders.

Now, consider the other case when P_u request for becoming a leader has been sent down the ring. In this case, P_v's request will never be forwarded. It is a lower priority request and the request will basically be dropped. As a result, P_v's request will not propagate down the ring and it can never be elected as a leader.

Hence, we have proven in both cases that leader election will always lead to one leader. Note that there is no starvation because the request of the highest priority process (lowest id) is bound to succeed. No other process will absorb it.

Message Complexity

To compute the worst case, we need to find a case where the maximum number of requests are sent. We want each request to travel as far (as many hops) as possible. Consider an arrangement of processes in chronological order. This means that process P_u sends a request to process P_{u+1} and then

process $\texttt{P}_{\texttt{u+1}}$ sends it to process $\texttt{P}_{\texttt{u+2}}$, so on and so forth. Finally process n sends its request to process 1.

Let us assume that all the processes are interesting in becoming a leader and they operate in lockstep. This is a synchronous computing model, which is fine. All synchronous settings are also asynchronous settings, not the other way around.

Say at $t = 1$, all the processes have their leader election requests ready. They send it down the ring to their neighbors. At $t = 2$, assume that all the requests have reached the neighbors. Other than the request from node n, which has reached process 1 and will get absorbed (not transmitted any further), the rest of the requests are not absorbed. This is because they are at processes that have sent requests that have a lower priority. Hence, the processes are bound to forward them to their neighbors. Ultimately, all the requests will reach node 1 at which they will get absorbed.

We can easily count the number of hops that all the requests traverse. The request from process n traverses just one hop $n \rightarrow 1$. The request from process $\texttt{P}_{\texttt{u}}$ traverses $n - \texttt{u} + 1$ hops. The total number of hops traversed or the number of messages sent is as follows:

$$M_{tot} = \sum_{i=1}^{n}(n - i + 1) = \frac{n(n+1)}{2} \tag{5.1}$$

We thus need to send $O(n^2)$ messages to elect a leader. For any process, the maximum number of messages sent is n (process 1 in this case). An overhead of $O(n^2)$ is not acceptable when we have a large number of processes. Let us thus design a more efficient solution.

5.7.2 Algorithm with $O(nlog(n))$ Message Complexity

Consider Algorithm 21. It makes similar assumptions as the Chang Roberts algorithm. The processes are organized as a ring and multiple leader election requests can be made simultaneously. The request with the lowest process id wins. However, in this case, all the requests need not traverse the full ring. We run many *mini algorithms* – local leader election algorithms in arcs of increasing sizes on the ring. This ensures that unless a process has a chance of winning the leader election, its request does not go very far.

Let us start with the INITIALIZE method. In this case, two $\langle$`probe`$\rangle$ messages are sent to the left and right neighbors. A $\langle$`probe`$\rangle$ message contains the `id` of the current process and two more parameters `k` (log of the search distance) and `d` (distance), which are set to 0 and 1, respectively.

The algorithm operates in a series of phases. `k` is set to 0 for the first phase, 1 for the second phase, so on and so forth. We assume that the leader election request ($\langle$`probe`$\rangle$ message) goes $2^{\texttt{k}}$ hops in both the directions (left and right) in a given phase. If it is not *absorbed*, the algorithm proceeds to the next phase – `k` gets incremented by 1. `d` maintains a count of how far the request has been sent in a given direction (left or right). It will never be allowed to exceed $2^{\texttt{k}}$.

The *field of view* of a process is limited to $2^{\texttt{k}}$ hops to its left and right for a given value of `k`. A phase succeeds if a process has the highest priority in its field of view. This further means that the leader initiation request (`probe` request) was not *absorbed* by any process in the field of view. We can thus think of the initiating process as the leader of all the processes in its field of view. Given that the field of view increases by a factor of 2 in every phase, ultimately one arm of it it will encompass the entire ring. This is when the initiating process will receive its own $\langle$`probe`$\rangle$ message assuming it has the highest priority. It will thus be elected as the leader. Let us now do a deep dive into the pseudocode shown in Algorithm 21 before we mathematically prove that this algorithm is indeed more efficient.

Algorithm 21 An optimized leader election algorithm

```
1:  procedure INITIALIZE
2:      send ⟨probe, id, 0, 1⟩ to left and right
3:  end procedure
4:  procedure RECEIVEPROBE(⟨probe, j, k, d⟩ from left(right)) ▷ The current process is the leader if my id (id) = j (id in the message)
5:      if j = id then
6:          leader ← id
7:          terminate()
8:      end if
        ▷ Keep propagating the message
9:      if j < id and d < 2^k then
10:         send ⟨probe, j, k, d + 1⟩ to right (left)
11:     end if
        ▷ Send a reply back
12:     if j < id and d = 2^k then State send ⟨reply, j, k⟩ to left (right)
13:     end if
        ▷ Absorb the request if none of the aforementioned conditions hold
14: end procedure
15: procedure RECEIVEREPLY(⟨reply, j, k⟩ from left(right))
16:     if j ≠ id then
17:         send ⟨reply, j, k⟩ to right(left)
18:     else
19:         if received ⟨reply, j, k⟩ from right(left) then
20:             send ⟨probe, id, k + 1, 1⟩ to left and right
21:         end if
22:     end if
23: end procedure
```

Consider the RECEIVEPROBE method defined in Line 4. A process call this method when it receives a leader election request from its left or right neighbor. Assume that it gets the request from process j. Consider the simplest case first when the process gets back its own leader election message (j = id, Line 5). In this case, we can just declare the current process to be the leader and terminate the algorithm.

Next, consider the general case where j < id. In this case, the process making the request (j) has a higher priority as compared to the current process. Hence, there is a need to propagate its request. Of course, if a leader has already been elected, then the request from j can be absorbed.

We are not showing that case in the interest of readability.

Assuming that no leader has been elected, if $\texttt{d} < 2^{\texttt{k}}$, then there is a need to propagate the request further. If it came from the left neighbor, it needs to be sent to the right neighbor, and vice versa. Every time the message is forwarded, `d` is incremented. It is like a running count.

Once $\texttt{d} = 2^{\texttt{k}}$ (Line 12), it means that the end of the current phase has been reached. There is a need to send a ⟨`reply`⟩ message in the reverse direction (towards the initiating process). If $\texttt{j} \neq \texttt{id}$, there is nothing much to do. The processes simply need to forward the reply message. However, if $\texttt{j} = \texttt{id}$, then there is some additional work to do. Note that two ⟨`probe`⟩ messages are sent in the left and right directions. We need to wait till replies come from both the sides. This would indicate that no leader election request has been sent by a higher priority process (lower id). Once both the replies are received, the initiating process becomes the leader in its field of view. It can now double the size of the field of view by incrementing `k`. It needs to then send a pair of ⟨`probe`⟩ messages to its left and right neighbors (shown in Line 20).

Proof

Consider the case when the highest priority process (lowest id) is the initiating process. Its messages will never get absorbed and its field of view will keep increasing. Ultimately, the termination criteria shown in Line 5 will become true and the algorithm will terminate after declaring a leader.

Let us now prove that it is not possible for two or more leaders to be elected simultaneously. Assume that processes with ids `i` and `j` are elected as leaders. Let $\texttt{i} < \texttt{j}$ without loss of generality. It is obvious that when `j`'s ⟨`probe`⟩ message reaches `i`, it will be absorbed, whereas the reverse will not happen. Hence, it is not possible to elect the process with `j` as a leader.

We will thus always have one leader elected as per this algorithm.

Message Complexity

Let us first consider the message complexity from the point of the view of the process that will be elected as a leader. `k` will vary from 0 to $log(n)$, where n is the total number of nodes. For every hop that a ⟨`probe`⟩ message traverses in one direction, it traverses one hop in the opposite direction as well. There are two associated replies as well. Hence, the total number of messages is:

$$\begin{aligned} 4 \times \sum_{k=0}^{log(n)} 2^k &= 4 \times (2^{log(n)+1} - 1) \\ &= 4 \times (2n - 1) \\ &= 8n - 4 \end{aligned} \tag{5.2}$$

The leader thus requires $8n - 4$ messages to be sent over the network. However, we also need to count the number of messages that the rest of the processes send. Here the assumptions matter. Let us assume a synchronous algorithm where we treat every phase as a round.

- The maximum number of winners after k phases is as follows:
  - Two winners need to at least be 2^k entries apart.
  - Thus, the total number of winners after k phases is $n/(2^k + 1)$
- The total number of messages for each initiator in phase k is 4×2^k
- Hence, the total number of messages in the k^{th} phase is:

$$4 \times 2^k \times \frac{n}{2^{k-1} + 1}$$

- The total number of messages is:

$$M = \sum_{k=0}^{log(n)} 4 \times 2^k \times \frac{n}{2^{k-1} + 1} = O(n\ log(n)) \tag{5.3}$$

We thus observe that the total number of messages that is sent is $O(n\,log(n))$ as opposed to $O(n^2)$ in the Chang Roberts algorithm. In both cases, the process that is going to be elected as a leader causes $O(n)$ messages to be sent.

5.7.3 Tree-based Leader Election

Instead of a simple ring-based overlay, we can look at other kinds of overlays as well such as tree-based overlays. Trees have higher connectivity than rings and thus algorithms based on trees often have reduced message complexities. A tree is also a very convenient structure for broadcasting messages. For

Algorithm 22 Waking up all the nodes prior to tree-based leader election

```
1: procedure INITIATE(Process u)
2:     if u is an initiator then
3:         awake ← true
4:         for each v ∈ neigh(u) do
5:             send wakeup to v
6:         end for
7:     end if
8: end procedure
9: procedure RECEIVEWAKEUP
10:    while numWakeups < | neigh(u) | do
11:        receive(wakeup)
12:        numWakeups ← numWakeups + 1
13:        if awake = false then
14:            awake ← true
15:            for each v ∈ neigh(u) do
16:                send wakeup to v
17:            end for
18:        end if
19:    end while
20: end procedure
```

example, if we want to broadcast the id of the leader who was elected, it can be done very easily with a tree.

To describe tree-based leader election, we have two algorithms. Algorithm 22 is run before the actual leader election algorithm starts. Here, the aim is to first wakeup all the nodes in the tree such that they can participate in the process of leader election. For a deeper explanation of tree-based algorithms, the reader can refer to the book by Gerard Tel [113].

Let us first focus on the method INITIATE. Note that we will continue to follow the same convention. `u` is the current node and all the other nodes are referred to as `v`. If a given process `u` is an initiator, then we set its state to `awake` and send a $\langle$`wakeup`$\rangle$ message to all its neighbors on the tree.

Processes call the method RECEIVEWAKEUP to receive and process all their $\langle$`wakeup`$\rangle$ messages. They run a *while* loop for η iterations, where η is the number of neighbors that process `u` has. When a process receives a $\langle$`wakeup`$\rangle$ message, if its `awake` variable is set to **false**, then it is set to **true** and the $\langle$`wakeup`$\rangle$ message is then forwarded to all the neighbors.

We are basically broadcasting the $\langle$`wakeup`$\rangle$ message to the nodes of the tree here. To minimize the traffic and avoid flooding, if a process has received the $\langle$`wakeup`$\rangle$ message once, then it is not woken up again.

Let us now consider Algorithm 23 for actual leader election. Let us first define two state variables $\texttt{min}_\texttt{u}$ and `received`. The former variable records the id of the highest priority process whose leader election request has been received by the current node (`u`). The other variable `received` records the number of leader election messages received up till now.

Each node (referred to as `u`) calls the method RECEIVEALL. Every node waits to get messages from each of its children. From a given child `v`, node `u` receives a message in each iteration. Let us refer to the id of the leader node in the message be `w`. Furthermore, we record the fact that a message has been received from `v`. Process `u` has an array $\texttt{rec}_\texttt{u}$ that has one entry for each child. It is initialized to **false**. The moment a message is received from process `v`, $\texttt{rec}_\texttt{u}[\texttt{v}]$ is set to **true**. We the increment the variable `received` and set the value of $\texttt{min}_\texttt{u}$ to $\texttt{min}(\texttt{min}_\texttt{u}, \texttt{w})$. The latter action basically means that we always set the smallest received id as the id of the current leader. Once the leader election message has been received from each child, the smallest id received ($\texttt{min}_\texttt{u}$) is sent to the parent node.

Algorithm 23 Leader election in trees

```
1: received ← 0
2: min_u ← u
3: procedure receiveAll
4:     while received < #children do
5:         receive ⟨w⟩ from v
6:         rec_u[v] ← true
7:         received ← received + 1
8:         min_u ← min(min_u, w)
9:     end while
10:    send min_u to parent
11: end procedure
12: procedure receiveFromParent(Leader w)
13:    if w = u then
14:        state ← leader
15:    else
16:        state ← lost
17:    end if
18:    for each v ∈ children(u) do
19:        send w to v
20:    end for
21: end procedure
```

Ultimately, the requests reach the root of the tree. The minimum id is the id of the leader. Given that every node in the tree was woken up prior to the leader election phase, every node got a chance to stand in the election. Once the root gets all the requests from its children, it can compute the minimum id (including its own) and find the leader. The last phase is to broadcast the id of the elected leader to all the nodes in the tree.

This is easily achieved by calling the method receiveFromParent. Each process u gets the id of the computed leader w from its parent node. If u = w, the current node u is the leader. It can set its state to leader otherwise it can set its state to lost (lost the leader election process). Next, we forward the message to each v ∈ children(u). When a message is received with the id of the leader, each process calls the method receiveFromParent.

The overall algorithm is thus quite simple. The leader election requests traverse up the tree. Given that every process is woken up and gets a chance to stand in the leader election, the root sees a comprehensive picture of the entire leader election process. All that it needs to do is compute the lowest

process id that it has received. That is the id of the leader process. It just needs to be broadcast down the tree. The correctness of this algorithm is quite obvious. We ask every process and ultimately filter out the lowest process id (among all the processes that are interested in getting elected as the leader).

A tree with n nodes has $n-1$ edges, and a message traverses an edge only once in a phase. There are three different phases of the algorithm: broadcast the $\langle$`wakeup`$\rangle$ message to all nodes, accumulate the leader election requests and broadcast the id of the computed leader. Each phase requires $n-1$ messages (one message sent on one edge) and there are three such phases. Thus, the message complexity is $3(n-1)$.

5.8 Distributed Spanning Tree: Gallager, Humblet and Spira (GHS) Algorithm

Given a set of distributed nodes, computing the minimum spanning tree is a very important problem. It is an important primitive that can be used to implement many other algorithms. For instance, a tree can be used to broadcast values to all processes or conduct a tree-based leader election. As a result, the distributed minimum spanning tree problem is very important. Let us discuss the famous Gallager, Humblet and Spira algorithm [45, 113] in this section.

5.8.1 Basic Results regarding MSTs

Before we design the algorithm, let us look at some basic properties of MSTs. Let us first prove a theorem regarding the uniqueness of MSTs.

Theorem 30 *Assume that each edge of the graph has a unique weight. There is a unique MST for the graph.*

Proof: Assume this statement is false. This means that there must be two different MSTs T_1 and T_2 with the same cumulative weight.

Let the edge e be in T_1 but not in T_2. Let us add the edge e to T_2. This will create a cycle. Assume that there is an edge e' in the cycle in T_2, which is not present in T_1. There is bound to be such an edge otherwise the same cycle will be present in T_1. Given that T_1 is a tree this can never happen. Now, given that edges have unique weights, the weight of e is not equal to the weight of e'.

There are two cases.

Case I: ($e < e'$) We can always remove e' from T_2 and add edge e. This will lead to an MST whose weight is less than T_2. This is not possible because it is assumed that T_2 has the least overall weight.

Case II: ($e' < e$) Let us do the reverse in this case. Let us remove edge e in T_1 and replace it with e'. We will end up with a tree with lower cumulative weight than T_1 (which is a MST). This is not possible.

Hence, in both cases we arrive at a contradiction. Hence, there will be one, unique MST. □

Let us define the notion of a *fragment*. It is a sub-tree of an MST. Consider all the *outgoing* messages from a fragment. An outgoing edge is defined as an edge that has one node in the fragment and one node outside it. A least weight outgoing edge (abbreviated as LWO edge) is the edge with the least weight among all outgoing edges of a fragment. Note that there will be only one such edge because all the edge weights are unique. Let us now prove a simple lemma.

Lemma 31 *Assume a fragment F. Let e be its LWO edge. $F \cup e$ is a fragment.*

Proof: Assume $F \cup e$ is not a fragment. Given that F is a fragment, let us assume that $F \cup e'$ is a fragment. Given our assumption, $e \neq e'$.

Let us now add edge e to the MST that contains fragment F. This will create a cycle given that e is not a part of the MST. This cycle will have edge e' because that also connects F to the rest of the tree. From this cycle, we can always replace e' with e. The resulting structure will still remain a tree because we have the same number of edges and the structure is connected. Given that the weight of e is less than the weight of e' owing to the fact that e is the LWO edge from F, this tree will have a lower weight than the assumed MST. This is not possible. Hence, we arrive at a contradiction.

Thus, we prove that $F \cup e$ is a fragment. □

5.8.2 Overview

Initially, each process (node in the tree) is a fragment. The fragments gradually grow in size incorporating neighboring nodes and edges. We can also merge fragments of a roughly similar size. This process of enlarging and merging fragments happens in parallel (concurrently) until the process terminates when only one fragment is left. That fragment is the MST.

An astute reader will agree that the Prim's algorithm [31] works conceptually in a similar manner. Initially, we start with a one-node fragment.

The fragment is gradually enlarged, one node at a time until the fragment cannot be grown any further. This is the MST. It is important to take into account that the edge that we add to the fragment is as per Lemma 31 – it is the LWO edge between the fragment and the nodes in the rest of the graph. The Prim's algorithm has an elaborate mechanism for finding such an edge. A min-heap is maintained to find this edge. However, in this case, we are adding one node at a time (via the LWO edge). The algorithm is thus sequential in nature, which militates against the ethos of pure distributed computing.

In this case we want a parallel algorithm that operates on the nodes of the tree concurrently. We can always find the LWO edge of a fragment concurrently, add it to the fragment, and grow it by one node. However, that is not enough. We need to look for more ways to enhance parallelism. A truly parallel algorithm should not add one node at a time – it should take much larger steps. Typically, we want such algorithms to run in $O(log(n))$ time. This is possible when we have a tree-structured computation where we merge sub-trees (fragments) of the same size and keep doing the same. We thus need to store the approximate size of a fragment and we need to be able to merge two fragments of the same size to create a larger fragment that is roughly twice the size of the original fragments. If we can do this frequently enough, we can achieve MST creation in approximately $O(log(n))$ time.

We implement this as follows. Each fragment has a name and a level. Consider two fragments F_1 and F_2. If $level(F_1) < level(F_2)$ and fragment F_1 wants to combine with F_2, we allow this to happen. In this case, all the nodes in F_1 take up the name of F_2. This is like a smaller fragment completely getting absorbed in a bigger fragment.

If $level(F_1) = level(F_2)$, then we can merge both the fragments as equals. We increment the level of the nodes in F_1 and F_2 by 1. The nodes in the merged fragment $F_1 \cup F_2$ take up a new name. Let us formalize this into two combining rules. The context is set in Figure 5.5 where there are two fragments F_1 and F_2. Assume that some node in F_1 initiates the process of merging. Let L_1 be the level of F_1 and L_2 be the level of F_2. Let the LWO edge from F_1 be e_{F_1} and from F_2 be e_{F_2}, respectively. If $L_1 = L_2$ and $e_{F_1} = e_{F_2}$ then it means that two equi-leveled fragments want to merge. Their LWO edges are the same. This means that we can happily merged the two fragments F_1 and F_2 and create a larger fragment $F_1 \cup F_2$. In this case, we increment the level of all the nodes in the merged fragment by 1. The name of the new merged fragment is the id of the common LWO edge $(e_{F_1} = e_{F_2})$.

The third case, which is when a fragment with a larger level wants to

merge with a fragment with a lower level is not allowed. A large fragment cannot gobble smaller fragments. We will see that this can effectively sequentialize the algorithm. Hence, we allow merges only between equals (fragments with the same level) or when a smaller fragment (lower level) with a larger fragment (higher level). There are two rules that we will regularly use in the rest of the algorithm: Rule LT (less than) and Rule EQ (equal to).

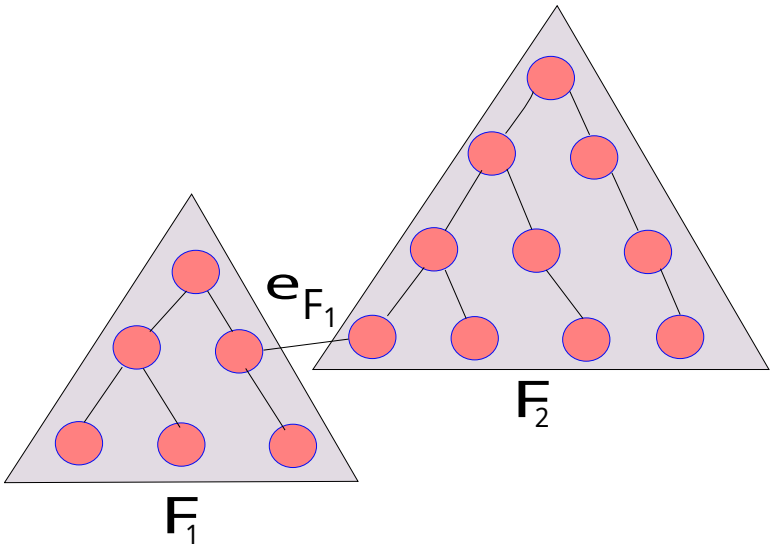


Figure 5.5: Two fragments with a common least weight edge

Rule LT If $L_1 < L_2$, then we combine F_1 and F_2. All the nodes in the new fragment have have the name of F_2 and level L_2.

Rule EQ If $L_1 = L_2$ and $e_{F_1} = e_{F_2}$, the two fragments combine with the following conditions.

- The level of all the nodes in the merge fragment $F_1 \cup F_2$ is $L_1 + 1$.
- The name is set to the LWO edge e_{F_1}.

5.8.3 The Algorithm

Messages and Data Structures

Each node has three states: asleep, find and found. It starts with the quiescent asleep state. Once the protocol starts, each node enters the find state

and then starts finding the LWO edge. Once the LWO edge is found, it transitions to the found state and then tries to combine using the *EQ* or *LT* rules. After the fragment has combined, it is time for the bigger fragment to find the LWO edge once again. Every node in the merged fragment enters the find state.

Each edge in the graph of nodes is tagged with a status. Initially, all the edges are marked basic. This means that their status is not determined as yet. Once an LWO edge of a fragment is found, we know that it is a branch in the MST (as per Lemma 31). Sometimes, the algorithm tries to connect with another node that already belongs to the partially constructed MST. This edge will induce a cycle. Hence, it is marked as reject.

Each node has the following fields: name, level, parent and a bunch of temporary variables. The parent in this case is a complicated concept as we shall see later. The temporary fields maintain bookkeeping information such as the best edge and node found up till now. These are required to compute the LWO edge of the fragment.

Pseudocode

Algorithm 24 shows the initialization procedure. Here, the convention is that the current node is `u` and any other node is represented as `v`. Let $\texttt{u} \rightarrow \texttt{v}$ (`uv`) be the least weight outgoing edge from `u`. Given that every node is a fragment in itself, by Lemma 8.9 `uv` is a branch in the tree. Hence, we set the status of the `uv` edge as branch. The level of each node is initialized to 0 and the `state` of node `u` is set to found. The number of received messages `rec` is set to 0. This is the initial state of each node. Subsequently, each node sends a ⟨`connect`⟩ message to `v`. `u` sends its current level 0. The reason that a ⟨`connect`⟩ message is sent is because `v` needs to know that `uv` is `u`'s LWO edge and thus `v` has been connected to the MST. Once a node `v` receives a ⟨`connect`⟩ message, it invokes the RECEIVECONNECT procedure.

Algorithm 24 The initialization procedure

```
1:  procedure INITIALIZE
2:      ▷ uv is the LWO edge from u
3:      status[v] ← branch                                  ▷ Edge in the MST
4:      level ← 0
5:      state ← found
6:      rec ← 0
7:      send ⟨connect, 0⟩ to v
8:  end procedure
9:
10: procedure RECEIVECONNECT(receive ⟨connect, L⟩ from v)
11:     if L < level then                                   ▷ Combine with rule LT
12:         status[v] ← branch
13:         send ⟨initiate, level, name, state⟩ to v
14:     else if status[v] = basic then
15:         WAIT ()
16:     else if L = level and status[v] = branch then       ▷ Rule EQ
17:         ▷ uv is the combining edge
18:         send ⟨initiate, level + 1, uv, find⟩ to v
19:     end if
20: end procedure
```

Let us assume that node u receives a ⟨connect⟩ message from node v whose level is L. The first case considers the case where L is less than the current level of u. In this case a smaller fragment wants to merge with a larger fragment. The smaller fragment can then be absorbed using the *LT* rule. In this case, the status of the uv edge is set to branch. Once, the connection has been established, it is important to let the smaller fragment know that its request for connection has been accepted. The level of all the nodes in the smaller fragment needs to change to that of the larger fragment and the name should also change to that of the nodes in the larger fragment.

Now, let us consider the rest of the two cases. In Line 24, we consider the case where we have an asymmetry: the least weight edge from v to u is not the same as the least weight edge from u to v. This is indicated by the fact that the status of the edge will be basic at the time of checking. This can either indicate that the current node has not completed the initialization process or an asymmetry indeed exists. In either case, rule *EQ* is not applicable. If the current node (u) has not completed the process of discovering which edge is its fragment's LWO edge, then that process needs

to complete first. Assume that the `uv` edge is indeed found to be the LWO edge. Then, a ⟨`connect`⟩ message will be sent; however, this time the status of the node at `v` will not be basic, it will be branch, and thus the process of merging will succeed (subject to the levels).

The condition for merging is applicable in Line 24, which we enter if the levels of both the fragments are the same. In this case, we can apply rule *EQ*. An ⟨`initiate`⟩ message is sent on the edge `uv` with an incremented level. The status find is also sent because it is a message to the other fragment that after it is merged, it needs to participate in the process of finding the LWO edge. This edge is known as the *combining edge*.

When an ⟨`initiate`⟩ message is received, the procedure RECEIVEINITIATE (Algorithm 0) is invoked. This basically means that a neighboring node has accepted a ⟨`connect`⟩ message because of either rule *LT* or *EQ* and it is responding in the affirmative.

The initiating node sends its current level, name and state ($\texttt{level}', \texttt{name}', \texttt{state}'$, respectively). The level, name and state of the current node are set to the respective values of the neighboring node (respectively) in (Line 25 of Algorithm 0).

Now, the aim is to propagate this information to the rest of the nodes of the fragment.

Algorithm 25 Receipt of the initiate message

```
1: procedure RECEIVEINITIATE( ⟨initiate, level′, name′, state′⟩, v)
2:     ▷ Set the state
3:     (level, name, state) ← (level′, name′, state′)
4:     parent ← v                          ▷ Set the parent to the initiator
5:
       ▷ Propagate the update
6:     bestNode ← φ
7:     bestWt ← ∞
8:     testNode ← φ
9:     for each r ∈ neigh(u): (status[r] = branch) ∧ (r ≠ v) do
10:        send ⟨initiate, level′, name′, state′⟩ to r
11:    end for
12:
       ▷ Find the LWO edge
13:    if state = find then
14:        rec ← 0
15:        FINDMIN()
16:    end if
17: end procedure
```

Every node needs to have a parent. It is initialized to the node itself. However, the way the parent is subsequently set is quite interesting (see Line 25). In this case, given that u got the ⟨initiate⟩ message from v, v is set to u's parent. If we are combining with the *LT* rule then it is clear that a sub-tree is becoming a part of a larger tree. The more interesting case is the *EQ* rule. In this case, the LWO edge has to be the same for both the fragments. If the LWO edge is edge uv, u is set to v's parent and vice versa. In fact, after initialization, this will continue to be the case for all fragments whose sub-fragments were ever created with the *EQ* rule.

The next step is to propagate the update that the fragment's name and level have changed down the subtree. Prior to doing so, we set a few node-specific state variables: bestNode (node in the fragment that is one end of the LWO edge), bestWt (weight of the LWO edge found up till now) and a testNode (we will test and see if a fragment merge can be done). For each neighbor of u (other than v), we forward the ⟨initiate⟩ message.

Once this has been done, then it is time to call the procedure FINDMIN, whose job is to find the LWO edge of the fragment.

Algorithm 26 The findMin method

```
1: procedure findMin
2:     if ∃v ∈ neigh(u): status[v] = basic, (w(uv) is minimal) then
3:         testNode ← v
4:         send ⟨test, level, name⟩ to testNode
5:     else
6:         testNode ← φ                          ▷ No more neighbors left
7:         processMin()
8:     end if
9: end procedure
```

Here again, the current node is `u`. We look at all the neighbors of `u` and find the edge with the least weight as long as its status is basic. Let us assume that there is such a node `v`. We set it to the `testNode`. Then, we send a $\langle$`test`$\rangle$ message to it. If this is not the case, then we record the fact that no such `testNode` was found. We then report the same by calling the processMin method.

The method receiveTestMsg (see Algorithm 0) is invoked when a node receives a $\langle$`test`$\rangle$ message. Along with the status of the message test, the other arguments are the id of the message sender `v`, its name and level. The latter two are `name`$'$ and `level`$'$, respectively.

If `level`$' >$ `level`, it means that `v` has a higher level. We do not allow such kind of a merge as per the EQ and LT rules. Thus, there is no option but to wait.

Let us consider the rest of the cases where the level of `v` is less than or equal to the level of `u`. If both have the same name, then it clearly means that the message was sent from one node of the fragment to another. It is thus an internal edge. We set its status to reject (cannot be an edge of the MST) if it was hitherto unlabeled (the state was basic).

Let us now draw our attention to Line 8. In this case, we test whether `v` is equal to `testNode`. We are basically checking if there is any outstanding $\langle$`test`$\rangle$ message that has already been sent to the `v`. If they are not equal, we send a $\langle$`reject`$\rangle$ message to `v`. The reason for this is that if they are equal then `v` is the `testNode` and it would have gotten a $\langle$`test`$\rangle$ message from the current node (`u`). `v` would inevitably realize that `uv` is an internal edge and it would mark the status of the edge as reject. There is no need for sending any additional message. If `v` = `testNode`, then we can proceed with the findMin method. This method will in turn find the next LWO edge and send it a $\langle$`test`$\rangle$ message.

The last case that we need to consider is when the names of both the fragments of the nodes `u` and `v`, respectively, are not the same. In this case, the edge `uv` is acceptable. This not a fragment's internal edge. It is the LWO edge from `u` to nodes in other fragments. We need to accumulate all such edges and compute the LWO edge for the entire fragment. Given that `uv` edge has been found to be acceptable, we send an ⟨`accept`⟩ message to `v`. This tells `v` that the edge `uv` is acceptable (not an internal edge and the *LT* and *EQ* rules can potentially be applied).

```
 1: procedure receiveTestMessage(test, level′, name′, v)
 2:     if level′ > level then
 3:         wait()
 4:     else if name = name′ then                    ▷ This is an internal edge
 5:         if status[v] = basic then
 6:             status[v] ← reject
 7:         end if
 8:         if v ≠ testNode then
 9:             send ⟨reject⟩ to v
10:         else
11:             findMin()
12:         end if
13:     else
14:         send ⟨accept⟩ to v                       ▷ different fragment
15:     end if
16: end procedure
```

Whenever a node receives a ⟨`reject`⟩ message (method receiveReject in Algorithm 27), it simply marks the status of the edge as reject and invokes the findMin method again until it is either successful or runs out of edges. If it is successful, it receives an ⟨`accept`⟩ message and calls the method receiveAccept (see Algorithm 27).

In this case, when an ⟨`accept`⟩ message is received from `v`, we set the `testNode` to `v`. Given that the edge `uv` has been found to be acceptable, we can compare with the best weight (`bestWt`) found up till now. If it is smaller, then we set `bestNode` to `v`. This is then reported by calling the processMin method.

Algorithm 27 The receiveReject and receiveAccept methods

```
1: procedure receiveReject(v)
2:     if status[v] = basic then
3:         status[v] ← reject
4:     end if
5:     findMin()
6: end procedure
7:
8: procedure receiveAccept(v)
9:     testNode ← φ                                  ▷ Reset the testNode
10:    if w(uv) < bestWt then
11:        bestWt ← w(uv)
12:        bestNode ← v
13:    end if
14:    processMin()
15: end procedure
```

The code of the processMin method is shown in Algorithm 28. A node waits to get a message from all its children (in the fragment). This happens as follows. `cnt` is a temporary variable that represents the number of children in the fragment. `rec` is a node-specific variable that is incremented when a message with the best-weight edge is received from a child. This will be discussed later when we talk about the receiveReport method. If `rec` = `cnt`, it means that a message regarding the least weight edge has been received from each child.

We also test if `testNode` = ϕ, which means that the current node has finished its least weight edge search process. If both the conditions are true, then we can set the state to found. This node is done. The `bestWt` can be reported to the parent by sending the `report` message to the parent node (Line 5).

Algorithm 28 The processMin method

```
1: procedure processMin
2:     cnt ← | {v : status[v] = branch ∧ v ≠ parent } |
3:     if (rec = cnt) ∧ (testNode = φ) then
4:         state ← found
5:         send ⟨report, bestWt⟩ to parent
6:     end if
7: end procedure
```

The receiveReport method's code is shown in Algorithm 29. There are two arguments: the node sending the message (v) and the LWO edge in its subtree (its weight is ω). There are two cases. The first case is when v is not the current node's parent.

In this case, we just update the bestWt variable if $\omega <$ bestWt and set bestNode to v because that is the node from which the information about the best weight came from. We also increment rec and then call the processMin method. There is a need to call this method after every message is received from a child node. Once we get messages from all the children and the current node has explored its edges, we send a report message to the parent (part of the processMin method).

Algorithm 29 The receiveReport method

```
1:  procedure receiveReport(v, ω)
2:      if v ≠ parent then
3:          if ω < bestWt then
4:              bestWt ← ω
5:              bestNode ← v
6:          end if
7:          rec ← rec +1
8:          processMin()
9:      else
10:         ▷ v = parent
11:         if state = find then
12:             wait()
13:         else if ω < bestWt then
14:             changeRoot()
15:         else if (ω = bestWt = ∞) then
16:             stop()
17:         end if
18:     end if
19: end procedure
```

Now, let us consider the next case when v is the parent of the current node u (Line 10). A node will receive a message from its parent only if it is a *combining edge* (see the description of the method receiveConnect). In this case, if the state is find, then it means that the current node is not done with searching for the least weight edge itself. There is a need to wait. Let us consider the other case where the current node is done with finding

the LWO edge in its subtree.

Otherwise, we compare ω with `bestWt`. If $\omega <$ `bestWt`, then it means that the LWO edge is in the subtree rooted at `v`. We can thus proceed to change the root of the tree by calling the method changeRoot. Note that in this case the root of the tree for the entire fragment will be set to the node of the LWO edge (in the fragment). This is the role of the method changeRoot.

It is of course possible that no best weight edge is found because the tree is fully formed. In this case $\omega =$ `bestWt` $= \infty$. At this point, the algorithm can stop – the MST has been computed.

Algorithm 30 The changeRoot method

```
 1: procedure changeRoot
 2:     if status[bestNode] = branch then
 3:         send ⟨changeroot⟩ to bestNode
 4:     else
 5:         ▷ Proceed along the LWO edge
 6:         status[bestNode] ← branch
 7:         send ⟨connect, level⟩ to bestNode
 8:     end if
 9: end procedure
10:
11: procedure receiveChangeRoot
12:     changeRoot()
13: end procedure
```

The pseudocode of the changeRoot method is shown in Algorithm 30. If `status[bestNode]` is `branch` then it means that we have not arrived at the LWO edge yet. There is a need to thus send this information towards the LWO edge (starting with `bestNode`). We thus send the message ⟨`changeroot`⟩ to `bestNode`. It receives this message and calls its changeRoot method (see Line 12).

Now consider the other case when `status[bestNode]` is not `branch`. In this case, we have arrived at the LWO edge. Given that the LWO edge from any fragment is a part of the MST (see Lemma 31) we can set the value of `status[bestNode]` to `branch`. This edge is a part of the MST now. We can initiate the process of connecting the fragment that contains the fragment that contains `bestNode`. It will be added as a part of the *LT* or *EQ* rules at some point of time.

5.8.4 Proof

We always find the LWO edge of a fragment. Every node in the fragment finds the LWO edge on its own. It verifies that the other end of the LWO edge is not in its fragment, and after it receives an ⟨`accept`⟩ message from it, it reports the id of the edge to its parent. All the nodes in the fragment collect this information and relay this information to the root nodes. The job of a root node is to compute the global minimum and decide the LWO edge.

Subsequently, the fragment tries to combine using the LWO edge. The request waits for some time and finally the fragment combines with either the LT or EQ rules. As per Lemma 31 this is how a fragment grows. Note that we are continuously making bigger and bigger fragments. Ultimately, this process will stop when the MST has been built.

5.8.5 Message Complexity

Let us make an assumption. Given a level k, assume that the fragment contains at least 2^k nodes. This trivially holds true when $k = 0$. When there is a combination with the LT rule, the level does not change. Hence, our assumption still holds true. Now consider combinations using the EQ rule. Let us assume by induction that the assumption holds. Let their levels be equal to k. The total number of nodes in the merged fragment (with level $k+1$) is at least 2^{k+1}. Hence, the assumption still holds. Using mathematical induction we can say that the results hold for all k. If $k = log(N)$, the size of the fragment will at least be N. The fragment becomes the MST. We thus observe that per node, there will be a maximum of $O(log(N))$ level changes. For all the nodes, the maximum number of level changes will be limited to $O(Nlog(N))$.

Let us now count the number of messages in a different way.

Consider the edges. A node is rejected only once (for a given edge). A ⟨`test`⟩ message is sent and the response is a ⟨`reject`⟩ message. Thus, a total of $2E$ messages (at most) are sent in this phase.

Let us now count the successful messages. Consider the number of messages a node sends or receives for a given level. At the most, we receive a single ⟨`initiate`⟩ message, receive an ⟨`accept`⟩ message, send a ⟨`report`⟩ message, send a ⟨`changeroot/connect`⟩ message and a ⟨`test`⟩ message. Thus the total number of messages in this category is limited to $5Nlog(N)$.

Hence, the total message complexity is $O(2E + 5Nlog(N))$.

5.9 Synchronizers

In the prior section, we have seen the design and analysis of multiple distributed graph algorithms. All of these algorithms work in the synchronized model of distributed computing. As you may recall, one of the fundamental properties of the synchronized model is that there is a notion of a round wherein all the messages sent in the prior round would reach their destination by the beginning of the current round. Further, the model does not handle link or node failures. On the other hand, the real-world is not operating in such a synchronized manner. If it were to, then the physical time for each round could be so long to allow for the longest link latency or computational delay to finish that the execution time of algorithms will be no longer practical. We still continue to keep the assumption that there are no failures of the nodes or the links. With this assumption, it implies that in the asynchronous model, any message sent will be eventually received but messages can get arbitrarily delayed and reordered in transit.

It appears therefore that the algorithms designed so far will not have practical use and one has to actually design algorithms that work in the asynchronous model of computation. Question **??** asks you to design an asynchronous algorithm for obatining a spaning tree (forest) of a graph using ideas similar to those of the synchronous BFS algorithm presented in Section **??**. Such native asynchronous algorithms are not easy to design for all problems.

However, we note that the two models that we are talking about, the synchronous model and the asynchronous model, are two ends of a spectrum of possibilities. One is very idealistic and the other is very pessimistic. In reality, it is worthwhile to understand the in-between space where the solutions work in practice and are also possible to design and analyze.

From an algorithm designer point of view, it will be good to explore alternative mechanisms that address the design of asynchronous distirbuted graph algorithms.

One mechanism that is often useful in this context is that of *simulation*. The idea is to create a way that an algorithm designed in one model can be made to run as an algorithm in a different model such that on the same input, both the algorithms produce the same output. Awerbuch [**?**] proposed the idea of *synchronizers* that allow for a simulation of a synchronous algorithm in an asynchronous setting. In the rest of this section, we will describe how to design the synchronizer mechanism.

5.9.1 The Simluation

At a high level, the idea of the synchronizer simulation is to create a mechanism at every node that allows the node to safely move from one round to the next round. This allows the node in the asynchronous algorithm to simulate the actions of the node in the synchronous algorithm in the next round.

This mechanism is known as a *pulse* that is generated at every node of the network. In addition, every message that is sent is piggybacked with the round number of the sender node. With these two steps, any synchronous algorithm can then be simulated in the asynchronous model as the following lemma shows.

Lemma 32 *Consider a synchronous algorithm $\mathcal{A}$ that runs in T_s rounds. If all nodes genreate safe pulses, then $\mathcal{A}$ can be simulated in the asynchronous model.*

Proof: Suppose that node v is the node that terminates at the end of the T_s rounds in algorithm $\mathcal{A}$. In the asynchronous model, once node v generates the (safe) pulse corresponding to round T_s, node v would have received all the messages it needs to perform its local computation and produce the required output as in $\mathcal{A}$ and terminate. □

The tricky question that the above lemma does not address is to define and characterize when to issue the pulse and when it is safe to do so at a given node. In the rest of this section, we discuss mechanisms to characterize when it is safe for a node to generate the pulse. The notion of safety is captured in the following definition.

Definition 33 *We define a pulse for round i at node v to be safe if the pulse is generated after (i) v generated all pulses for all rounds j such that $j < i$, and (ii) v received all the messages that were sent to v by any nodes in rounds j with $j < i$.*

Based on the above definition, let us reword the questions for simplicity as follows. Consider the situation that all nodes already sent the pulse for a round i, $i \geq 1$, then we want to know when any node v can generate the pulse for round $i + 1$. The main challenge however for v is to know when node v can conclude that all messages intended for v and sent in a round i or before have been received at v. Notice that v does not know if some neighbor of v sent a message for v in a prior round. In the absence of such information, it is not clear if v has to wait for a delayed message from a

neighbor or no such message is due at v. In such a situation, if v waits for too long and possibly an infinite amount of time to generate the pulse, this inaction from v may result in potential deadlocks. On the other hand, if v decides too soon that it is no longer expecting any messages sent in the prior rounds, then v advances in error and the pulse thus generated is not safe.

One way to resolve this dilemma is to require a small modification of the synchronous algorithm wherein every node sends a message to every one of its neighbors in every round. Let $M_i(u, v)$ denote the message that node u sends to its neighbor v in the ith round. If u does not send a message to v in a paritcular round or node u terminates its execution by round i, we set $M_i(u, v) = \perp$.

This discussion leads to our first simple synchronizer called the α-synchronizer. The details of this synchronizer are given in the following theorem.

Theorem 34 *Let $\mathcal{A}$ be a synchronous that runs in T_s rounds. Uisng the alpha-synchronizer, we can simulate $\mathcal{A}$ in an asynchronous system so that the algorithm runs in time T and uses an additional $O(T|E|)$ messages compared to $\mathcal{A}$.*

Proof: Our proof shows that each node can generate safe pulses for all the rounds it executes in. In particular, the following statements all hold:

- The pulses generated by all nodes are safe.
- Node v sends the message $\langle M_i(v, w), i\rangle$ to each of its neighbors w in round i.
- If node v terminates in a round $t_v \leq T$ according to $\mathcal{A}$, then all the neighbors of v receive all the messages sent by v for all rounds 1 to t_v. Further, over rounds t_v to T, the neighbors of v recieve the dummy message from v.
- When a node v terminates, it is able to produce the same output according to $\mathcal{A}$.

We start by mentioning when a node generates a pulse. A node v generates a pulse for round i only after it receives messages from all of its neighbors that were sent in rounds prior to i. (See also Question 5.8). It is easy to see that pulses generated accordingly are safe.

The above statements can be shown by using induction on the round number. Since each node sends a message TODO

For the bound on the number of messages, we observe the following. In each of the T rounds that the simulation runs, each link carries a message in each direction: either the message according to $\mathcal{A}$ or the dummy message ($\perp$). Thus, the number of messages across the T rounds is $O(T|E|)$. □

We notice from Theorem 34 that while it is possible to simulate a synchronous algorithm in an asynchronous system, we pay a price in terms of the number of messages needed in the simulation. This problem is more acute if the synchronous algorithm $\mathcal{A}$ sends very few messages over its T rounds, then the simulation does incur a heavy overhead. An alternative mechanism, called the β-synchronizer, remedies this situation as described below.

5.9.2 The β-synchronizer

Notice that the $O(|E|)$ messages per round of simulation is due to the condition that each edge carry a message in every round of the simulation. It is possible that many of these messages are not part of $\mathcal{A}$, the synchronous algorithm, but are introduced by the simulation. Let us call the messages introduced by the simulation as control messages. The purpose of these control messages is to aid nodes in generating safe pulses. Reducing the number of control messages is a way to reduce the total message count. To this end, we create a model where the control messages are carefully orchestrated.

This orchestration combines elements of the upcast and downcast subroutines that we developed in Section **??**. In particular, we consider a rooted spanning tree H of the network G on which a synchronous algorithm $\mathcal{A}$ is operating. The control messages are sent only along the edges of H. The roof of H initiates and collects the control messages via a downcast operation and subsequently uses an upcast operation to send and receive control messages to all nodes in H. These downward and upward control messages act to indicate nodes when they can move to a subsequent round. We start with the following definition and provide further details in the following.

Definition 35 *A node v is safe with respect to a pulse if all the messages during the execution of $\mathcal{A}$ that v sent in that pulse have been received at their destination.*

In the β-synchronizer based simulation, the root of the tree H initiates the pulse when it is safe for the root node. Using Definition 35, the root can identify if it is safe as follows. Since there are no failures to nodes or links, all the messages sent are eventually received. The receiving nodes send an

acknowledgement to indicate the receipt of the message. When the root node receives all the required acknowledgements, it can deduce its safety.

The pulse message generated by the root node is then forwarded to the children of the root node. Each such child of the root waits until it is past itself is safe and the pulse message from the root arrives to propagate the pulse to its children. It also marks its state as safe. The maximum time between which a node sends two consecutive pulses constitutes an asynchronous round. These pulse messages travel downward to the leaves of the tree akin to the downcast operation.

Once the leaves recieve the pulse message, they send a safe message to their parent. An internal node that is in the safe state, upon receiving safe message from all of its children, sends a safe message to its parent. Once the root node receives a safe message from all of its children, it is safe for the root to conclude that one round of $\mathcal{A}$ is successfully simulated and attempt the generate the next pulse for the next round.

The above discussion leads us to the following theorem on the β-synchronizer.

Theorem 36 *Let $\mathcal{A}$ be a synchronous algorithm that works in T_s rounds and send $M_{\mathcal{A}}$ messages. Let H have a depth of d_H from the root to some leaf node. It is possible to simulate algorithm $\mathcal{A}$ in $O(d_H T_s)$ asynchronous rounds using $O(M_{\mathcal{A}} + nT_s)$ messages.*

We have seen two mechanisms for simulating a syncrhonous algorithm in an asynchrous model. One of them uses a large number of messages but takes less time, whereas the other reduces the number of messages but incurs a significant time overhead. One question that is interesting in this context is to see if there are mechanims that get us the best in terms of both the extra message overhead and the extra time overhead. The γ-synchronizer does exactly that.

5.9.3 The γ-synchronizer

The reason that the α-synchronizer uses more messages is due to requiring that each edge carry a message in every round. In a similar fashion, the reason that the β-synchronizer rquries more time is due to attempting a global synchronization drein from the root of a spanning tree on top of the network. If we aim to reduce the number of messages and the time together, one possibility is to attempt an α-synchronizer style synchronization within small sized clusters, and to attempt a β-synchronizer across clusters. Creating such clusters is the goal of algorithms for network partitioning, first

introduced by Awerbuch and Peleg [**?**] and are further studied in several other works, viz [**?**].

The details of the γ-syncrhonizer using a partitioning are as follows. Let us consider the situation where the network G is partitioned into clusters such that each:

- Each node belongs to exactly one cluster
- Each cluster C has a diameter of at most $d*$.

We build the following two structures on top of the clusters. Firstly, we obtain a tooted spanning tree T_C in each cluster C. From the property of the clustering, the tree T_C has a depth of at most $d*$. Secondly, suppose there exist edges of the form uv such that u and v are in distinct clusters C_1 and C_2. In this case, we pick one such edge xy as the intercluster edge between C_1 and C_2. This edge xy will serve as the only communication link for control messages between C_1 and C_2. Let E_X denote the set of intercluster edges. The situation at the end of these two steps is shown in Figure **??**.

The basic idea is to run the β-synchronizer within each cluster and the α-synchronizer across the clusters using the intercluster edges. For the former, we notice from Section **??** that the additional time needed using the β-synchronizer is now $O(T_{\mathcal{A}}d*)$ and for the former we use no more than $O(T_{\mathcal{A}})$ rounds. So, the overall time needed is in $O(T_{\mathcal{A}}d*)$. The former uses $O(M_{\mathcal{A}})$ whereas the latter uses $O(T_{\mathcal{A}} \times (E_X + n))$ messages. Put together, we have the following theorem.

Theorem 37 *Let the network G have a paritioning such that each partition has a diameter of $d*$. Let $\mathcal{A}$ be an algorithm that runs for $T_{\mathcal{A}}$ rounds using $M_{\mathcal{A}}$ messages in the synchronous model. Then, $\mathcal{A}$ can be simulated in an asynchronous setting using $O(T_{\mathcal{A}}d*)$ time and $O(M_{\mathcal{A}} + T_{\mathcal{A}} \times (E_X + n))$ messages.*

5.10 Further Reading and Summary

Coloring, MIS, etc: Connection between spanners and compact routing

Lower bounds? We did not mention much of these in the chapter – connect to communicatio complexity book of Noam Nissan.

Questions

Question 5.1. Rewrite Algorithm 16 so that the number of iterations is possibly reduced.

Question 5.2. Justify the following with respect to Algorithm 17. For every graph G, there exists an order of choice of the vertex $v$in Line 3 such that the MIS obtained is of the largest size.

Question 5.3. Will the above statement be true when the algorithm is used for obtaining a maximum independent set?

Question 5.4. Write the algorithm of Baswana and Sen for constructing a 3-spanner as a distributed algorithm. Also, mention the size of messages sent in the algorithm.

Question 5.5. Write Algorithm 17 so that it works in a distributed setting using conventions from Section **??**.

Question 5.6. In the context of the dominating set algorithm discussed in Section 5.4.4, show that if a node u is good, then the probability that node u is covered in a given round is at least $1 - 1/\sqrt{4}e$.

Question 5.7. In the context of the dominating set algorithm, let m denote the maximum rounded span of any node at the start of a round. Let us call each element in cover(v) with rounded span m as an *entry*. Show that in every round, at least one third of the entries are good, top entries.

Question 5.8. In the proof of the simulation corresponding to Theorem 34, will the following rule for node v to generate the pulse for round i work?

> **Rule:** Node v generates the pulse for round i once v receives all the messages sent by its neighbors in round $i - 1$.

Justify your answer considering that the simulation operates in an asynchronous setting.

Chapter 6

Consensus and Agreement

6.1 Formal Definition of the Consensus Problem

In the world of distributed systems, if there is one fundamental question that is the cornerstone of almost all the major algorithms and designs, it is the *consensus problem*. This problem is known by many names. It is often referred to as the *agreement problem* as well. The basic idea is very simple. There are n processes (independent entities). Each one of them proposes a value. Let us say that process i proposes value v_i. in some generic versions of the problem, it is not necessary that a process propose a value. It can instead decide to remain quiescent. After the protocol starts, the processes start communicating with each other via either sending messages or communicating by other channels such as shared memory. At the end of the protocol, which is guaranteed to terminate, we want that all the processes to *agree* on a single value, which one of the processes has proposed. For example, they cannot all agree to the value zero, if it is not been proposed by any process. Hence the caveat is that the value that is agreed to by all the processes is a value that at least one process has proposed.

Let us now further complicate this problem. Often in a distributed system, we can have faulty processes. In fact, if we do not have faulty processes, most algorithms in this area become very simple and trivial. Hence, almost all realistic distributed systems take faulty processes into account. Faulty processes can be various types. We can have simple fail-stop processes that simply become unresponsive when they fail. Furthermore, we could have processes that are intermittently faulty. This means that they are active for some time, then they become inactive, then they can become active and so

on. This appears to be a very innocuous kind of fault model because the processes cannot cause a lot of damage. A process can become unresponsive and not participate in the protocol but it looks like it cannot be capable of damaging the overall system in other ways. Let us compare this with another class of faults known as *Byzantine faults*. In this case, a process can genuinely act maliciously. It can deliberately lie and provide different inputs to different processes. Furthermore, a set of malicious processes can collude (act together) and deliberately misguide the rest of the processes. Such Byzantine faults are in many ways more powerful than regular fail-stop faults; they make the process of agreement quite difficult in some settings such as systems where the processes share a common clock.

Definition 38 *The consensus problem is defined as follows. We have a set of processes, where each process may propose a value. Finally, when the protocol terminates, one of the values is accepted by all the non-faulty processes. This value is known as the consensus value, which must have been proposed that at least by one of the processes.*

Hence, there is a need to slightly modify our definition. We need to say that when the protocol terminates, all the non-faulty processes agree on the same value. We do not really care what the faulty processes agree on because given the fact that they are faulty, their conclusions do not matter. Insofar as we are concerned, we don't take their actions very seriously after the protocol terminates.

A second question that arises is whether the value that is finally agreed upon by all the non-faulty processes needs to be proposed by a non-faulty process. Can it be proposed by a faulty process, which is possibly guilty of duplicity and has possibly communicated different values to different processes? The answer lies in the fact that it is not possible for a process to ascertain for sure whether another process is faulty or not. It needs to take whatever message that it gets at face value unless it runs a different protocol to ensure that a given process is faulty. Hence, it is a wise idea to not impose any restriction on the value that is agreed upon as long as it has been proposed by some process and the same has been communicated to at least one non-faulty process. Definition 38 captures these augmentations to the basic definition.

Let us now consider an example and see why this problem is so important. Often it is the case that while we are booking an airline ticket, we require multiple entities to be in sync. These are the credit card company, the mobile app, the backend server (the travel company) and the airline.

Sometimes it does happen that money is deducted from the credit card, however the ticket has not been booked. This can be a very irritating and annoying situation. Sadly, this has happened to one of your authors on many occasions. The reason that this happened in the first place is because there was no agreement or there was no consensus between all of these interacting parties. Hence, it was possible for the money to be deducted from the credit card account, without any ticket being actually booked. Of course in this case, the backend server of the travel site did realize that there is a mistake and the money was refunded in due course of time. But sadly, the author was not aware of whether a ticket has actually been booked or not, and this caused a lot of delay and confusion. Finally, when the ticket was booked in the second attempt, the prices had increased !!! Had a consensus protocol been operational, this would not have happened.

In this chapter, we will start with some classic impossibility results, which basically say that in many situations of practical interest, it is not possible to design an algorithm that guarantees reaching a consensus all the time. In fact, we shall see a very surprising result in Section 6.2 that says that if we have one faulty process in an asynchronous system (without a common clock), it is impossible to achieve consensus in a distributed setting. As disappointing as this news may be, we shall pile on bad news on top of bad news and introduce another result that basically says that in any system that is prone to network failures and partitions, consensus is not achievable unless we let go of a few more basic properties. This is known as the CAP Theorem. Even though, we cannot achieve consensus reliably hundred percent of the time, there are a lot of algorithms that help us achieve consensus most of the time. We will discuss many of these algorithms in an increasing order of complexity.

6.2 FLP Result

Let us prove one of the most important results in distributed systems that deals with the impossibility of consensus in an asynchronous setting with even one faulty process. It was proposed by Fischer, Lynch and Paterson in 1985 [42]. Based on the last names of the authors, this result is known as the FLP result. In this case, the faulty process can either be genuinely faulty, malicious or just slow. We will not have any way of finding out why a process is unresponsive. In this case, we further simplify the problem by considering binary consensus: 0 or 1. Our line of argument in this section is quite similar to the arguments made in the FLP paper, albeit with a

different method of explanation and modified terminology.

Every process is modeled as a finite automata. In a step it can either receive a message from another process, change its state upon receiving a message, or send a message to another process. Processes have broadcast capability. This means that if a message is sent to a non-faulty process, then it is guaranteed that all non-faulty processes will eventually receive it. However, messages can be delayed arbitrarily and also be delivered out of order. The proof relies on the fact that there is a small window of time where it is not possible to exactly ascertain if a process is slow, unresponsive or dead.

Let us consider a process p with input register x_p and output register y_p. A register is a region of storage that stores a single bit. Let the registers store one among the three values $0, 1, b$. The input registers contain the value that each process is proposing. Here, b is a special value that means uninitialized. All the output registers start with the value b. This means that we have not committed to an output. Once the output register finalizes to a value 0 or 1, it means that a decision has been made. The consensus value is either 0 or 1. The output register is "write once". This means that it can be written to only once. This means that once a decision regarding the consensus value has been made, it cannot be reversed. Let the **state of the entire system** be the union of the contents of the internal registers, output registers and internal states of all processes.

Let us assume that the processes send messages to a virtual message buffering system. This represents all the messages that are in flight. Let the term (p, m) be a tuple representing the destination process p and message m, where message m is meant to be delivered to process p. Of course, given that the network can have an arbitrary delay there is no guarantee that the message Will be delivered immediately or within a bounded amount of time. There are two basic primitives here: send (p, m), and receive (p).

The function send (p, m) means that we intend to deliver message m to process p. This tuple is internally placed in the message buffer and delivered at a later point in time. The function receive (p) deletes a message (p, m) from the message buffers and process p will receive the message m. If the message is not ready to be delivered and the destination process calls receive(p), then a null value (ϕ) is returned.

The configuration of the entire system is a combination of the internal states of all the processes and the contents of the message buffers. In the initial configuration C_{init}, the states of all the message buffers is empty and the states of all the processes is in the initial state. Let us assume that a process takes one step at a time: receive a message (could also be null), do

some internal computation and send messages to the rest of the processes. Let us refer to an *event* as a tuple of process and message (p, m). If an event is applied to a configuration C, $e(C)$ denotes the resulting configuration. A process can always be applied to an event (p, ϕ) and thus some forward progress is always possible. Assume $D = e(C)$, we can represent the same as $C \xrightarrow{e} D$.

Let us define a schedule of events that is a sequence of events that can be applied to a configuration, C. We will use the term σ to represent a schedule. $\sigma(C)$ represents a schedule where the end result is the final configuration (result of applying the schedule to configuration C). Let all the configurations of the form $\sigma(C)$ be called *accessible configurations* – they are reachable from the configuration C.

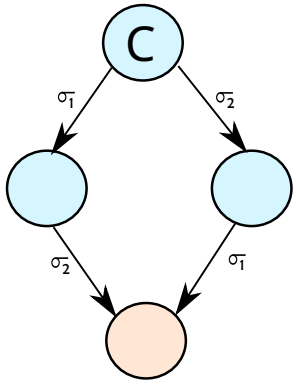


Figure 6.1: The commutativity of schedules if they do not have overlapping processes

We can quickly deduce a few common sense results. Assume that there are two schedules σ_1 and σ_2 where the processes that participate in them are disjoint. We claim that they are commutative. Consider a starting configuration C. We claim that $\sigma_1(\sigma_2(C)) = \sigma_2(\sigma_1(C))$. This is because the schedules are disjoint and as a result their order does not matter because the processes in the two different schedules do not interact. Figure 6.1 shows this graphically.

A few more definitions are due. A configuration C has a decision value v if some process has reached a decision state. This means that its output register y_p is equal to v. A consensus protocol is said to be partially correct if it satisfies the following conditions.

- Consider all the accessible configurations. None of them should have

more than one decision value.

- For each value v in the set $\{0, 1\}$, some accessible configuration has value v.

A *partially correct* protocol means that the protocol can lead to at least two kinds of configurations. One kind of configurations conclude 0 and the other type of configurations conclude 1. Of course, it is not necessary that a decision can be made all the time. There is a possibility that we get into an infinite loop and no decision is eventually made. The important aspect is that we never reach a configuration that is incorrect (two different values are decided) and it is possible that we can reach configurations that decide either value.

Next, a process p is said to be *non-faulty* if it takes infinitely many steps. This means that it never stops. Note that a process can always react to the ϕ message. It is otherwise faulty, which means that at some point it abruptly stops. A run (or execution) is said to be *admissible* if at most one process is faulty. The rest of all the processes are non-faulty, and they eventually receive every message. Note that the message buffers do not drop messages.

A run is a *deciding run* if a process reaches a decision state (output register is 0 or 1). Any partially correct consensus protocol is said to be *totally correct* if every admissible run is a deciding run. This means that a decision is made all the time. Recall that in the theory of complexity, we typically deal with such questions.

Now, the key question that we need to answer is as follows. Do totally correct protocols exist? Or in other words, do all partially correct protocols always have deciding runs? Note that we are only considering admissible runs here – at most one process is faulty.

The key FLP result says that it is not possible to design a totally correct protocol if there is one fault. Let us start with a sequence of lemmas to set the stage.

First, a few definitions are due. Let us say that a configuration is *univalent* if it only leads to configurations with a decision that is either only 0 or only 1. This means that for this configuration, a decision has already been made. Even if it hasn't been made, any reachable configuration with a decision value always decides the same value. On the other hand, a configuration that can reach configurations that decide either of the two values are called *bivalent*. This means that no decision has not been committed to. Things are open-ended at this stage.

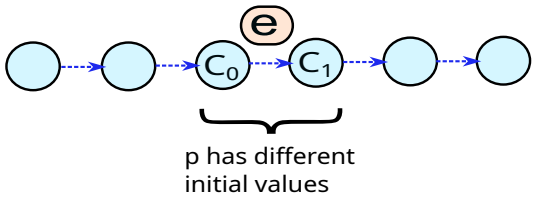


Figure 6.2: A path from a 0-valent to a 1-valent state. Note the positions of the configurations C_0 and C_1, and the event e that is associated with process p receiving the input to take it to C_1.

Lemma 39 *Every totally correct protocol has a bivalent initial configuration.*

Proof: Let us prove this by contradiction. If this is not the case, then all initial configurations are either 0-valent (decide only 0) or 1-valent (decide only 1). Two initial configurations can be termed to be *adjacent* if the only difference is in the initial values of some process p. For instance, the input register of one can be 1 and the other can be 0. Consider Figure 6.2. If we draw a path between two configurations that are 0-valent and 1-valent, respectively, then there will be two configurations on either side (one is 0-valent and the other is 1-valent). Given that these two configurations are adjacent, let the process p have differing initial values. Let the two configurations be C_0 and C_1, respectively. The former is 0-valent and the latter is 1-valent.

Consider a deciding run, which is admissible and process p does not take any steps. Given that we are considering a purely asynchronous network, this is indeed possible. Let the schedule be σ. Let the event associated with p taking a step to move from C_0 to C_1 be e. Now, let us consider the two sequences of events: $\sigma(e(C_0))$ and $e(\sigma(C_0))$. Given that in σ p does not take any steps and in e, p is the only process that takes steps, they are disjoint. Given that such operators are commutative we can easily conclude that $\sigma(e(C_0)) = e(\sigma(C_0))$.

Now, $e(C_0) = C_1$, by definition. Hence, $\sigma(e(C_0)) = \sigma(C_1)$. This is a run whose final outcome is 1 regardless of the subsequent steps (this follows from the definition of C_1). Let us consider the other term $e(\sigma(C_0))$. $\sigma(C_0)$ is a run whose ultimate outcome will be 0 (based on the definition of C_0). The same logic holds for $e(\sigma(C_0))$. We thus see that configuration C_0 can lead to two possible outcomes: 0 and 1. This leads to a contradiction. We had originally assumed that C_0 is a univalent configuration. However, that

did not turn out to be correct. C_0 happens to be a bivalent configuration.

We can thus conclude that any totally correct protocol will always have a bivalent initial configuration. □

Given that there is an initial bivalent configuration. The question is can we reach another bivalent configuration from it. If we can prove that we indeed can, then we are essentially laying the foundation for a proof where we can endlessly continue to have bivalent configurations. This means that we can never make a decision.

Lemma 40 *Consider a totally correct protocol. There is a maximum of one faulty process. Assume that C is a bivalent configuration. Consider all configurations of the form $\sigma(C)$. Consider the set of all configurations of the form $e(\sigma(C))$, where e is an event that is not a part of σ. There is a σ such that the configuration $e(\sigma(C))$ is bivalent.*

Proof: Let the set $\mathcal{C}$ be the set of of all configurations that are reachable from C without applying event e. Let the set $\mathcal{D} = \{e(C)|C \in \mathcal{C}\}$. We need to prove that $\mathcal{D}$ has a bivalent configuration. Let us assume that this is not true. This means that all the configurations in $\mathcal{D}$ are univalent.

Now, given that the configuration C is bivalent, there need to be two univalent configurations C_0 (decides 0) and C_1 (decides 1) that are reachable from C. Note that such configurations will always exist given that C is bivalent. Let us refer to the configurations as C_i, where $i \in \{0, 1\}$. If $C_i \in \mathcal{C}$, let us define $D_i = e(C_i)$. It is also possible that the event e was applied while reaching C_i from C, then there exists a configuration C_x such that the following sequence of events holds: $C \rightarrow \ldots C_x \xrightarrow{e} D_i \ldots \rightarrow C_i$. Recall that we are assuming that all D_i are univalent.

We have shown two cases. In the first case, $C_i \xrightarrow{e} D_i$ and in the second case $D_i \rightarrow \ldots \rightarrow C_i$. We thus see that the set $\mathcal{D}$ has both 0-valent and 1-valent configurations.

Now, let us look at two neighboring configurations in $\mathcal{C}$. We claim that we can find two such configurations E_0 and E_1 (both elements of $\mathcal{C}$) that satisfy the following properties. The first is that one is reachable from the other: either $E_0 \rightarrow E_1$ or $E_1 \rightarrow E_0$. Assume $E_0 \xrightarrow{e} D_0$ and $E_1 \xrightarrow{e} D_1$. D_0 and D_1 are univalent (decide 0 and 1, respectively). Let us prove that such a pair of configurations exist. Consider all pairs of univalent configurations in $\mathcal{D}$. For each pair of configurations in $\mathcal{D}$, consider the corresponding pair in $\mathcal{C}$. It is not possible that there is no pair where one configuration (in $\mathcal{C}$) is not reachable the other configuration (also in $\mathcal{C}$). If that is the case then there will be a strict partition between the configurations that lead to 0

and 1-valent configurations in $\mathcal{D}$, respectively. However, that is not possible because all configurations in $\mathcal{C}$ are reachable from the initial configuration.

We thus have the following set of relationships (without loss of generality).

$$\begin{aligned} E_0 &\xrightarrow{e'} E_1 \\ E_0 &\xrightarrow{e} D_0 \\ E_1 &\xrightarrow{e} D_1 \end{aligned} \tag{6.1}$$

Let us assume that the event e' happens for process p', and e happens for process p. Consider the following cases.

$p \neq p'$ In this case the events e and e' are commutative. We thus have $e'(e(E_0)) = e(e'(E_0))$. This implies that $e'(D_0) = e(E_1) = D_1$. One univalent configuration D_0 cannot lead to another univalent configuration D_1. This situation is shown in Figure 6.3

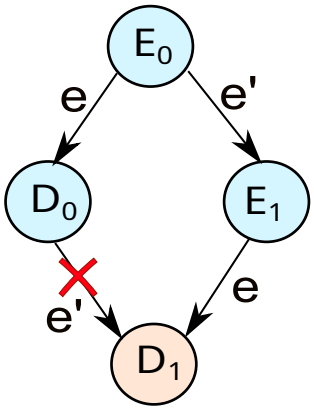


Figure 6.3: The illustration of Case 1 ($p \neq p'$)

$p = p'$ Consider a finite run from E_0 that is also deciding. Assume that the process p remains quiescent (does not take any steps). Consider such a run σ that ends up in a univalent configuration E_x. This situation is shown in Figure 6.4. In this case, the events e and e' commutate with σ. We thus have $e(\sigma(E_0)) = e(E_x) = F_0$. $e(e'(\sigma(E_0))) = \sigma(e(e'(E_0))) = \sigma(e(E_1)) = \sigma(D_1) = F_1$. F_0 and F_1 are univalent configurations that decide 0 and 1, respectively. If we look at

Figure 6.4, we observe that there is a path from the configuration E_x to F_0 and F_1. Now, the contradiction here is that there is a path from the univalent configuration E_x to two other univalent configurations F_0 and F_1 that decide two different values. There is a contradiction here also.

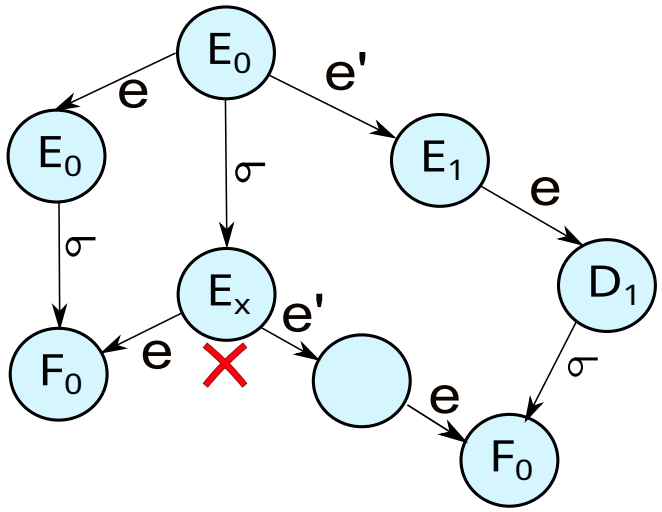


Figure 6.4: The illustration of the second case ($p = p'$)

This means that $\mathcal{D}$ does not have purely univalent configurations. It definitely has one bivalent configurations. □

Theorem 41 *It is not possible to design a totally correct protocol if there is one fault.*

Proof: Given the results of Lemma 39, we can conclude that there is a bivalent initial configuration C_b. This means that at this point, we have not decided whether the final outcome is 0 or 1. This is a fair assumption in any protocol that always terminates with a result as we have seen. This means that the result is not fixed a priori, which should be the case with consensus protocols. If the outcome has already been decided a priori, then there is no point of running the consensus protocol.

Now, let us assume that we have a network between processes where we assume that there is a message queue for each process. If a message needs to be sent from process p_1 to p_2, then all that p_1 needs to do is deposit a

message in p_2's message queue. The messages in each message queue are sorted based on the time at which they entered their message queue.

Let us consider the processes in a round-robin order. Given a process, let us consider the earliest message (part of event e) in each message queue. By Lemma **??**, there is a schedule in which e is applied at the end and we end up with a bivalent configuration. Using mathematical induction, we can argue that we will continue to have bivalent configurations regardless of the number of messages and events. This means that ultimately we will not be able to make a decision, because we will always end up with a bivalent configuration. □

The conclusion that we can derive from Theorem 41 is that if a process is slow, we may never be able to reach a deciding configuration. This means that the protocol may run forever without terminating. As a result, in an environment with at least one faulty process, it is impossible to design a consensus protocol that terminates all the time. However, in practice, a lot of consensus protocols can be designed that are simple, elegant and terminate *most of the time*. Just because, in all possible scenarios, it is not possible to guarantee termination does not mean that consensus protocols should not be designed and optimized.

6.2.1 Consensus with "Initially Dead" Processes

Assume there are n processes in the system. Let us slightly relax the assumptions here. Assume that no process suffers from any failure after the execution has started. This ensures that a lot of the problems that we saw in the case of a single faulty process shall not happen. There however could be *dead processes* that are unresponsive throughout the execution. Moreover, assume that a majority of the processes are not faulty. However, this information including the information regarding dead processes is not known to all processes. A process only knows about itself. It does not know whether another process is alive or not unless that process sends it a message.

The idea is to create a group of processes that mutually agree with each other. If the size of this group is large enough, then we can prove that a majority backs the consensus value. Once a value is decided upon, it cannot be changed in the future.

We will use the notion of the *quorum* here. A quorum in this case is a simple majority, which for n processes is $\lceil (n+1)/2 \rceil$ $(= Q)$ processes. The ceiling arises because the number of processes can both be even and odd. For example, if the number of processes is 7, $Q = 4$. This is a majority. If $n = 8$, $Q = 5$, again a majority.

We can design a 2-stage algorithm. In the first stage, every process sends its proposed value to $Q-1$ processes. The reason we have $Q-1$ here is because if we include the sending process, then we have Q processes. Every process does this. We could however have slow links, which may prohibit all messages reaching their destinations. Let us assume that a process waits to get $Q-1$ messages from other processes. This is a fair assumption even in an asynchronous setting. Given that processes shall not die in the middle, this will ultimately happen.

Once a process receives $Q-1$ first-stage messages, it enters the second stage. It sends a message to the processes that it got these $Q-1$ messages from, and asks them about all the messages that they have received. A process that gets a second-stage message waits till it gets $Q-1$ first-stage messages. Once that happens, it replies to all second stage messages with all the first-stage messages that it got. Each message contains the id of the sender and the proposed value.

At the end of the second stage, every process has a list of processes and the values that they have proposed. The process computes a union of the process lists. For process p, let this list be L_p.

Our claim is that $\forall p, p' : L_p = L_{p'}$. Assume that this is not the case. This means that there is a process $p_x \in L_p$ and $p_x \notin L_{p'}$. Process p_x sent messages to $Q-1$ processes. Let this set along with p_x be S_x ($\mid S_x \mid = Q$). All the processes in S_x are aware of p_x.

Assume that p' asked lists from $Q-1$ other processes. Clearly $p_x \neq p'$ and p_x is not in the list of processes that received messages from p'. Let this set be S' ($\mid S' \mid = Q$). None of the processes in S' are aware of p_x. Otherwise, p' would have known at the end of the second stage.

Let us look at the intersection $S_x \cap S'$. It is not null because their size is equal to Q, which is a majority. Hence, we can conclude that $S_x \cap S' \neq \phi$. The process that is a part of the intersection could not have died in the middle (this is not allowed as per our assumptions). The process that is a member of the intersection could not have been aware of p_x and not aware at the same time. Hence, we have a contradiction here. This situation is not possible.

We can thus conclude that $\forall p, p' : L_p = L_{p'}$. Given that all the processes have the same list of processes with them at the end of the second stage. They also have the list of proposed values (one per process). Given that all processes have exactly the same list of proposed values, any function that takes a list as input and returns a value is good enough for choosing the consensus value. Of course, the consensus value has to be a part of the list. That is a trivial requirement. One that is satisfied, all the processes come

to the same conclusion.

Hence, in this case, consensus is possible. Of course, the only caveat is that process failures are not allowed during the execution of the protocol.

6.3 Consensus in Synchronous Settings: Byzantine Agreement

Let us now consider solving the consensus problem in synchronous settings with Byzantine faults (see Section 1.2). In this case, we can make an assumption that a message, or a reply will be received in a bounded amount of time. If a process is down, this can be detected because it will be found that it is not sending messages when it is supposed to. Given that all the processes share the same clock, this is very easy to ensure. Most synchronous algorithms can be divided on the basis of rounds. In a round, a set of messages are sent and the messages are received. Based on the messages that a process receives, it updates its state accordingly and prepares messages to be sent in the next round. The absence of a message can thus easily be detected in a round. Moreover, we also assume that a message cannot be tampered with and cannot be deleted from the system. This means that the receiver is always sure who the sender of a message is and the fact that the message that has been received is the same that was sent.

6.3.1 Overview of the Problem

Let us consider the diagram shown in Figure 6.5. There is one commander and then there are multiple generals or *lieutenants*.

Consider a battle. The commander issues orders to his lieutenant generals. However, both the commander as well as his generals suffer from Byzantine faults. They can either be loyal or disloyal. Figure 6.5 shows a situation in which the commander is disloyal. He very smartly sends $\langle$`Attack`$\rangle$ (A) orders to two generals and $\langle$`Retreat`$\rangle$ (R) orders to the rest of the two generals. In this case, utter confusion prevails. The lieutenant generals do not know what to do and they are confused. A lieutenant general does not known what orders have been sent to the rest of the lieutenant generals. As a result, they are not sure how to act. There also could be another scenario where the commander is loyal and dutifully sends the same order to all lieutenant generals. However, some of the lieutenant generals themselves could be disloyal. They may decide to disobey the order.

Now, the question is how does a loyal lieutenant general distinguish

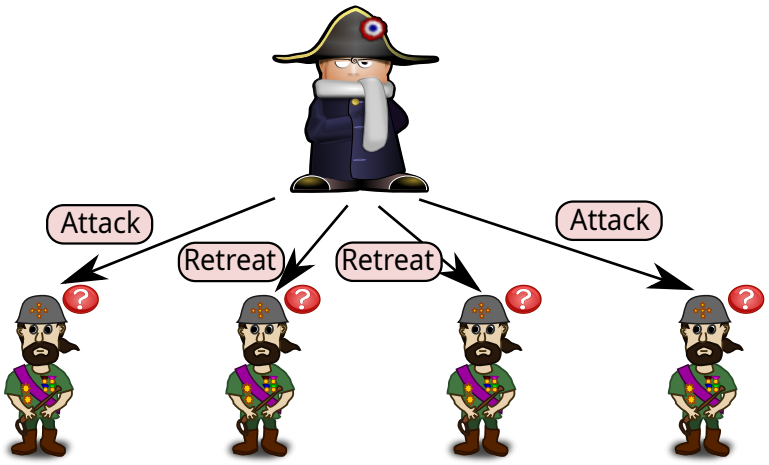


Figure 6.5: A Byzantine commander and generals

between these two cases? Both appear identical to him. He does not know whom to trust. One option is to of course ask the other generals about the orders that they have received. However, what is the guarantee that they are speaking the truth. There could be divergent answers. We thus have a complicated situation on our hands where a general does not know whom to trust.

If we want to create an effective protocol in such a scenario, we need to first define our objectives. Let us thus define two conditions.

Condition C1: All loyal lieutenant generals obey the same order.

Condition C2: If the commander is loyal, then every loyal general obeys the commander's order.

Definition 42 *In any Byzantine consensus problem, there is a commander and a set of generals – any subset of them can suffer from Byzantine faults. The commander is a process that sends a value to the rest of the processes for agreement. The rest of the processes are the lieutenant generals who don't know if the commander process is fault-free or not. The solution to the Byzantine consensus problem needs to satisfy two properties.*

1. *All the loyal generals follow or obey the same or order. This means that all fault-free processes arrive at the same decision.*

2. *If the commander is loyal, then all loyal generals follow the commander's order. This basically means that if the commander process is fault-free, then every fault-free process accepts the value sent by it.*

Note that this is a binary consensus problem, where we want all the loyal generals (including the commander) to agree on one of two values: ⟨`Attack`⟩ (A) or ⟨`Retreat`⟩ (R). Of course, this is not a pure consensus problem, where all the generals are equally privileged. The commander has a special place. If he is loyal, then his order is the consensus value.

6.3.2 Impossibility Results

Consider a situation with 3 generals: one commander and two lieutenant generals.

Theorem 43 *When we have one commander and two lieutenant generals and one traitor, the Byzantine consensus problem is not solvable.*

Proof:

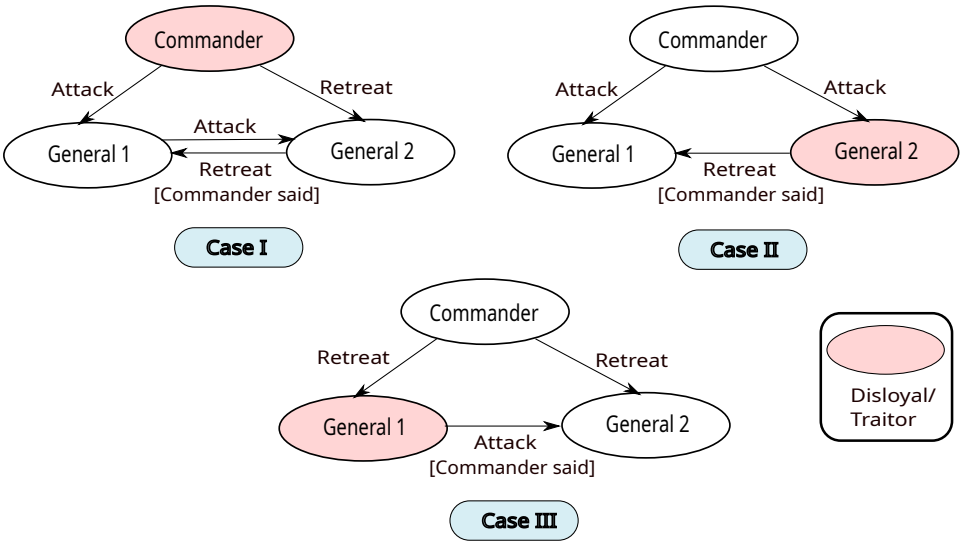


Figure 6.6: Three cases: disloyal commander (Case I), loyal commander sending ⟨`Attack`⟩ (Case II) and loyal commander sending ⟨`Retreat`⟩ (Case III)

We show tree cases in Figure 6.6. Consider cases I and II, and look at it from the point of view of General 1. He asks General 2 about the order

that he has received from the commander. In both the situations, General 1 receives ⟨`Attack`⟩ and ⟨`Retreat`⟩. General 1 cannot thus distinguish between the two cases. As per Condition C2, in the second case, General 1 should follow the order ⟨`Attack`⟩. Given that the first and second cases look identical to General 1, in the first case also, it needs to choose ⟨`Attack`⟩. This means that in Case I, **General 2 also needs to decide on** ⟨`Attack`⟩. Now, look at the third case.

In this case, the commander is loyal and sends the order ⟨`Retreat`⟩ to both the lieutenants. Hence, the order ⟨`Retreat`⟩ needs to be followed by General 2. From the point of view of General 2, the first and third cases look exactly the same. Hence, in the first case also, **it needs to decide to** ⟨`Retreat`⟩.

We thus have a contradiction here. General 2 cannot simultaneously decide to ⟨`Attack`⟩ and ⟨`Retreat`⟩. Hence, the consensus problem is not solvable here. □

The question that arises here is whether this result holds if we have m traitors out of $3m$ generals (including the commander). It turns out that it does. In fact there is a simple proof that is just a mere extension of Theorem 43.

Theorem 44 *Assume that there are a total of* $3m$ *generals (including the commander) and* m *traitors. It is not possible to solve the Byzantine consensus problem.*

Proof: Consider an instance of a regular Byzantine consensus problem with 3 generals, one of them being a commander. In a real setting, each general is a process.

Let us now create $3m$ separate processes, where each process acts like a general. On the lines of the proof in the original paper [], let us refer to these $3m$ generals as Albanian generals. Furthermore, let us assign m Albanian generals to one Byzantine general. This means that one Byzantine general process simulates m Albanian general processes.

The Albanian commander will be simulated by the Byzantine commander. The Byzantine commander will additionally simulate $m - 1$ Albanian generals. Now, given that only one Byzantine general is a traitor, this automatically translates to the fact that m Albanian generals are traitors, which is consistent with the statement of the theorem.

Let us now assume that we have an algorithm to solve the Byzantine consensus problems for the Albanian generals. Let us run this algorithm then. Each Byzantine general will need to simulate the actions that all of

its constituent Albanian generals (m in total) will take. Assume that the end result of the protocol is that we can solve the problem for the Albanian generals.

This means that the solution obeys both our conditions: C1 (all loyal generals obey the same order) and C2 (if the commander is loyal, then all the loyal generals obey the commander).

Both the loyal Byzantine generals that simulate the $2m$ loyal Byzantine generals can obey the same order that their simulated loyal Albanian generals obey. Given that all the loyal Albanian generals obey the same order, the Byzantine generals that are simulating them can also obey the same order. Let us assume that they do so. As a result, Condition C1 is satisfied for the simulating Byzantine generals as well.

Next, consider C2. If the Albanian commander is loyal, then all the loyal Albanian generals obey the order sent by the commander. If the Albanian commander is loyal, then it automatically implies that the simulating Byzantine commander is also loyal. Let it issue the same order. Hence, the Byzantine commander and all the m Albanian generals that it simulates (including the Albanian commander), all obey the same order. There will be another loyal Byzantine general that simulates m loyal Albanian generals. All those m loyal Albanian generals will obey the message sent by their commander because it is loyal. Let the simulating Byzantine general, who is loyal, also obey the same message. Hence, we observe that C2 holds for the Byzantine generals as well. We clearly do not care about the third Byzantine commander and all the m Albanian generals that it simulates because they are presumed to be traitors.

This means that using the solution for the problem of Albanian generals, we were able to solve the Byzantine consensus problem with three Byzantine generals where there is one traitor. This is clearly impossible and violates the results of Theorem 43. As a result, there is a contradiction.

Hence, it is not possible to solve the Byzantine consensus problem with Albanian generals. In other words, if we have $3m$ processes and m processes have Byzantine faults, then we cannot solve the Byzantine consensus problem. □

Theorem 44 is a powerful result. It basically says that no Byzantine consensus or agreement can be achieved if we have 33 faulty processes in 99 processes. An interesting question that can be posed here is what if there are 100 processes? In this case, we have m traitors and $3m+1$ generals. It turns out that we can. This is one of the most popular results in fault-tolerant distributed systems.

6.3.3 Consensus in the Presence of Byzantine Faults

Solution of the Problem with 4 Generals and one Fault General

Let us consider the simplistic version of the general problem where we have four generals (including the commander). At most one of them could be suffering from a Byzantine fault. We claim that there is a very simple solution to this problem. Let us assume that we have one of two cases (see Figure 6.7): the commander is a traitor and without loss of generality General 1 is a traitor.

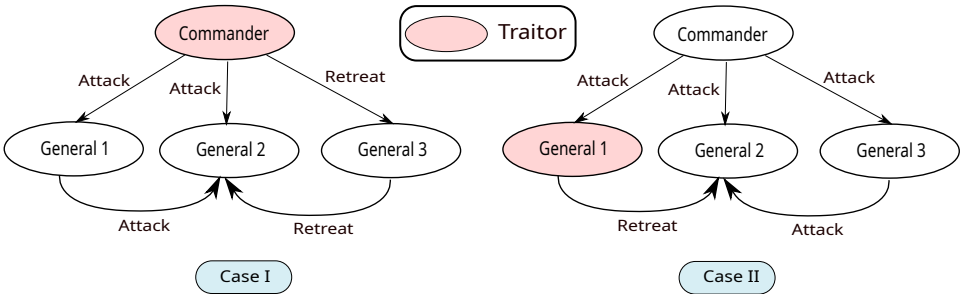


Figure 6.7: Case I: Commander is a traitor. Case II: General 1 is the traitor.

The algorithm is very simple. In the first round, the commander sends an order to each of the three lieutenant generals. Given that the loss of a message can be detected by the receiver, let us assume that a receiver assumes that it has received the ⟨`Retreat`⟩ message if no message arrives by the end of the round. In all cases, each receiver gets a message by the end of the round.

Consider Case I in Figure 6.7. The commander is a traitor and the lieutenants are clearly not aware of this fact. The commander sends its messages. In the next round, all the lieutenant generals send the message that they received from the commander in the previous round to the other lieutenant generals. If we look at all the messages that General 2 receives, then we have a set of three messages. It receives ⟨`Attack`⟩ in the first round and {⟨`Attack`⟩, ⟨`Retreat`⟩ } in the second round. The set of messages that it receives is thus: {⟨`Attack`⟩, ⟨`Attack`⟩, ⟨`Retreat`⟩ }. Let us assume a majority function that returns the most common value. If there are an equal number of ⟨`Attack`⟩ and ⟨`Retreat`⟩ messages, then we can choose ⟨`Retreat`⟩. Let us consider the set of messages that all the three lieutenant

generals receive and the majority value (see Table 6.1).

General	Messages	Majority value
General 1	{ ⟨Attack⟩, ⟨Attack⟩, ⟨Retreat⟩ }	⟨Attack⟩
General 2	{ ⟨Attack⟩, ⟨Attack⟩, ⟨Retreat⟩ }	⟨Attack⟩
General 3	{ ⟨Attack⟩, ⟨Attack⟩, ⟨Retreat⟩ }	⟨Attack⟩

Table 6.1: Set of messages received by each of the three generals

As we can see from the table, in this case all the three lieutenant generals receive the same set of messages and the majority function computes exactly the same result – ⟨Attack⟩. Condition C1 gets satisfied (all loyal generals obey the same order). Condition C2 is not relevant here because the commander is not loyal.

Let us now consider the second case in Figure 6.7. In this case the commander is loyal but General 1 is a traitor. Let us look at the messages that Generals 2 and 3 receive. Assume that General 1 told General 3 that it got the ⟨Attack⟩ message from the commander. Then, the respective sets of messages are {⟨Attack⟩, ⟨Retreat⟩, ⟨Attack⟩ } and {⟨Attack⟩,⟨Attack⟩, ⟨Attack⟩ }. In both the cases, the majority value is ⟨Attack⟩. As a result, all loyal lieutenants – Generals 2 and 3 in this case – obey the same order (Condition C1). This order is ⟨Attack⟩, which was sent by the loyal commander (Condition C2). Hence, we see that both the conditions are satisfied.

There are two important conclusions that we can draw here. The first is that it makes sense to create smaller groups in subsequent rounds and share the messages received in the previous round with them. For example, we created a number of smaller groups out of the lieutenant generals. Each group had a group leader (commander) that sent the message it had received in the previous round to members of the group. This is suggestive of a *recursive algorithm*. Second, we computed a *majority function* over of all the messages received in previous rounds. We showed that the majority values will be the same at the end of the protocol. Hence, both our conditions C1 and C2 got satisfied.

We can furthermore conclude that in this setting with at most one faulty general out of 4 generals, the Byzantine consensus problem is solvable. The only theoretical abstraction we are using is the fact that a recursive algorithm needs to be used where each lieutenant general in the previous round becomes the commander in the next round for a small set of generals. Sec-

ond, we relied on the majority function that was found to be stable and robust all kinds of scenarios.

Solution of the General Problem

Algorithm 31 Algorithm for Byzantine agreement

```
1: procedure runCommander(Value v, set of lieutenants ℒ)
2:     Send v to each lieutenant in ℒ
3:     If by the end of the round, a lieutenant does not receive a message,
       then it assumes it has gotten the ⟨Retreat⟩ message
4: end procedure
5: procedure runConsensus(m traitors, value v, set of lieutenants ℒ)
6:     if m = 0 then
7:         runCommander(v, ℒ)
8:         return v
9:     end if
10:    𝒱 ← ϕ
11:    for each Lieutenant i do    ▷ Treat lieutenant i as the commander
12:        ℒ′ ← ℒ − i
13:        𝒱 ← 𝒱 ∪ runConsensus(v, m − 1, ℒ′)
14:    end for
15:    return majority(𝒱)
16: end procedure
```

Algorithm 31 shows the algorithm for the general problem. The first procedure, i.e, runCommander is straightforward. In this procedure, we assume that there are no traitors. However, the commander can still go down without sending any message. It will never be the case that it will send some messages and fail for the rest. We assume that broadcasting a message to all $\mathcal{L}$ lieutenants is atomic (happens in one go). The commander sends its value v to all the lieutenants. Given that we are running a synchronous algorithm here, we have the notion of *rounds*. At the end of the round, if a lieutenant does not receive a message from the commander, then it can assume that it has gotten the ⟨Retreat⟩ message. Given that there are zero traitors, either all the lieutenants receive the same message or all of them do not receive a message and assume that they got the *Retreat message*. In the latter case, it is clear that the commander has gone down. In both the cases, all the axioms of Byzantine consensus hold.

Let us now look at the general case shown in the runConsensus procedure, where there are m traitors at most. Let the set of all the lieutenants be $\mathcal{L}$. If there are no traitors, we run the procedure runCommander. In this case, the value simply needs to be broadcasted to all the lieutenants in $\mathcal{L}$. No other step needs to be taken because we are assuming that there are no traitors ($m = 0$).

If $m > 0$, we have the general case. For each lieutenant `i`, we run a new instance of the procedure runConsensus, where we assume that lieutenant `i` is the commander and the set $\mathcal{L} - \texttt{i}$ are the lieutenants. This is a reduced instance of the problem, where there is one less lieutenant. We, in this case, assume that there is one less traitor, hence, we send $m - 1$ as the argument to runConsensus.

Let us broadly see what is happening here. If there are no traitors, and this fact is known, the algorithm is clearly correct. We simply run the runCommander procedure, where we simply needs to broadcast the value. Now, for each lieutenant `i`, all the loyal generals in $\mathcal{L} - \texttt{i}$ agree on the value that was received by `i`.

Let us assume that there are a total of n generals including the commander. In the first round, the commander sends its value to $n - 1$ lieutenants. In the second round, for each lieutenant `i`, we initiate a procedure for the rest $n-2$ lieutenants to agree on what value `i` received from its commander. Let us not care about how many rounds this process takes. Consider the point of time when this process concludes successfully. **At this point of time**, each loyal lieutenant agrees on what value was received by the rest of the lieutenants. Now, consider lieutenant `i` again. It has the value that it received from its commander and it has $n-2$ values that it believes the rest of the $n - 2$ lieutenants had received from the commander. This *belief* is a consensus belief in the sense that all loyal lieutenants share the same set of values. The claim is that the majority value of this set is the same for all loyal lieutenants (something that we need to prove). This is the consensus value of runConsensus, which is the return value for an instance with n generals and m traitors.

This algorithm is a classic recursive construction, where to solve a bigger problem, we solve smaller instances of the same problem. Ultimately, we reach the base case where there are no traitors and a simple broadcast suffices.

Proof of the General Solution

Let us now prove Algorithm 31. Let us abbreviate the function runConsensus $(m,\mathtt{v},\mathcal{L})$ as $RC(m)$, where m is the first argument. It corresponds to the maximum number of traitors in the runConsensus algorithm. Let us assume that the rest of the two arguments are known.

Lemma 45 *For any p and q, $RC(p)$ satisfies Condition C2 if there are more than $2q+p$ generals and at most q traitors.*

Proof: Let us have n generals. Clearly, $n > 2q+p$. In this case, we need to interpret p slightly differently. It corresponds to the number of times we shall make nested recursive calls.

Since we are evaluating Condition C2, which says that the order of a loyal commander is obeyed by all loyal generals, let us assume that the commander is loyal.

If $p = 0$, this is trivially satisfied. This is because the commander broadcasts its value to all the generals and the loyal ones obey him.

Assume it is true for $p-1$. Let us use induction to prove that this result holds for p. In the first round of the algorithm, the commander will send the value $\mathtt{v}$ to all $n-1$ lieutenants. Each loyal lieutenant will then call $RC(p-1)$.

We have the following result.

$$\begin{aligned} & n > 2q+p \\ \Rightarrow & n-1 > 2q+p-1 \end{aligned} \tag{6.2}$$

This means that runConsensus $(p-1,\ldots)$ will run correctly with the rest of our parameters (induction assumption). This further means that all loyal lieutenants who will be commanders now for their respective instances of the reduced problem ($RC(p-1)$), will broadcast the value $\mathtt{v}$ correctly.

If $p \geq 1$, then we have $n-1 > 2q$. This means that a majority of the lieutenant generals are loyal. Now, in $RC(p-1)$, a majority of the lieutenants receive the value $\mathtt{v}$ because each loyal general dutifully broadcasts $\mathtt{v}$ to the rest of the lieutenants. Given that they have also received $\mathtt{v}$ from their commander, the majority function returns $\mathtt{v}$. As a result, all the loyal lieutenant generals decide the value $\mathtt{v}$. This proves Condition C2 for $RC(p)$. □

Lemma 45 basically establishes the fact that the if the traitors are in a minority after subtracting p from the number of generals, then Condition

C2 holds. We were able to use majority-based arguments to prove that this holds because all the loyal generals will compute the same majority result. Let us now use this lemma to prove the main theorem.

Theorem 46 *$RC(m)$ satisfies both the conditions – C1 and C2 – when we have at most m traitors and at least $3m + 1$ generals (including the commander).*

Proof: Let us use induction to prove this result.

Consider the base case, i.e., $m = 0$. In this case, we have one general who is himself the commander. He clearly obeys his own order if he is loyal. This is thus a trivial case.

Let us now consider the general case. Assume that the theorem holds for $RC(m-1)$.

Assume that the commander is loyal. In this case, let us consider the results in Lemma **??**. Assume $p = m$ and $q = m$. Then the lemma stipulates that we have more than $2m + m$ $(=3m)$ nodes. The protocol follows C2. Consider now C1. This is trivially satisfied given that the commander is loyal and C2 holds (all the loyal lieutenants obey the commander's order).

Next, assume that the commander is not loyal. This means out of the lieutenant generals, at the most $(m-1)$ generals are traitors. When we run $RC(m-1)$, we have at least $3m-1$ generals (minus the commander in the previous round and the lieutenant initiating the protocol). We have $3m - 1 > 3(m-1)$. This means that we run $RC(m-1)$, we have more than $3(m-1)$ generals. As a result, the RC algorithm can be used with the argument set to $m-1$. It is covered by the induction hypothesis.

Now, all loyal lieutenant generals while solving $RC(m)$ initiate an instance of $RC(m-1)$. Each instance of $RC(m-1)$ further initiates instances of $RC(m-2)$, so and so forth. However, let us not bother about this recursive process. Let us assume that $RC(m-1)$ works correctly. Let us use this fact to prove that $RC(m)$ is also correct. Given that we have n generals that also includes the commander, we can assume that each of the $n-1$ lieutenant generals starts an instance of $RC(m-1)$.

Now consider all the instance of $RC(m-1)$. Each general stores the values that it agrees on in a vector. The vector has $n-1$ entries, where each entry corresponds to the consensus value of $RC(m-1)$ that was either initiated by itself or another lieutenant general. Note that contents of this vector are going to be the same for all loyal lieutenant generals. Given that $RC(m-1)$ works correctly, for each instance of $RC(m-1)$, all the loyal generals agree on the same value (Condition C1). Similarly, they have a consensus for the rest of the entries in their $n-1$ entry vectors.

Each vector V will be of the form $\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_{n-1}}$. $\mathtt{v_i}$ is the consensus value of $RC(m-1)$ initiated by lieutenant i. This is the greatness of this algorithm that because of Condition C1, which $RC(m-1)$ satisfies, the vectors will be identical for all loyal generals. As a result, a majority function computed on the vectors will always return the same result. If there are an equal number of ⟨Attack⟩ and ⟨Retreat⟩ entries, we can always arbitrarily choose either ⟨Attack⟩ or ⟨Retreat⟩. As long as this is a consistent choice, all the loyal generals will arrive at the same decision.

Hence, they satisfy Condition C1. Note that given that the commander is a traitor in this case, Condition C2 cannot be satisfied.

Hence, for all cases, Byzantine consensus holds. Hence, we prove that the runConsensus algorithm is correct as long as we have m traitors and at least $3m+1$ generals. □

Theorem 46 is a very important theorem. It basically says that the number of traitors have to be less than one-third the number of generals. Only then is a solution possible, we showed that it is not possible to find a solution to the Byzantine consensus problem. A simple majority does not suffice mainly because we cannot trust the commander. Otherwise, a simple majority would have been enough (see Lemma 45).

Message Complexity

Consider the problem $RC(m)$ with a total of n generals (including the commander). We send $n-1$ messages to $n-1$ lieutenant generals. Each lieutenant starts its instance of $RC(m-1)$. If the time complexity of $RC(m)$ is $T(m)$, then we have the following recurrence relation.

$$T(m) = (n-1)T(m-1) \tag{6.3}$$

The solution can be expanded as follows:

$$T(m) = (n-1)T(m-1) = (n-1) \times (n-2) \times (n-m) \times T(0) \tag{6.4}$$

In this case $T(0)$ is as simple as broadcasting the value to the rest of the $n-m-1$ generals. There is no need for running any additional protocol because we are assuming that there are no traitors. Equation 6.4 thus has a solution, which is as follows.

$$T(m) = (n-1) \times (n-2) \times (n-m) \times (n-m-1) = \frac{(n-1)!}{(n-m-2)!} \tag{6.5}$$

Hence, we see that the message complexity is indeed very large. In fact, if we use the Stirling's approximation [31], it can be shown that it is exponential in terms of the number of traitors. This is the price for finding a consensus solution with Byzantine faults. Clearly, if we have a few traitors like 1 or 2 traitors, then the cost of the solution is not much – it is either quadratic or cubic. However, the moment we have close to $n/3$ traitors, the message complexity blows up. Over the years, many algorithms have been proposed to speed up Byzantine consensus. However, the algorithms are still either quite slow or do not achieve a consensus all the time.

6.3.4 Solution with Signed Messages

Given the complexity of achieving Byzantine consensus in a general setting, consider the case where we use signed messages. In this case, a general can *sign* a message indicating that he is the originator or propagator of a message. The signature of a loyal general cannot be forged. Any attempted forgery can be detected. However, it is possible to forge the signature of a traitor and traitors can also collude in this process. In any case, the signature of any general (loyal or traitor) can be verified.

It turns out that if we have m traitors, we can find a solution for any number of generals. The notion that a solution can only be found out for more than $3m$ generals does not hold here.

Let us number the commander as General 0. A signed message looks like this: $\mathtt{v}{:}\mathtt{0}{:}\mathtt{g}_1{:}\mathtt{g}_2{:}\mathtt{g}_3$. Here, the value is $\mathtt{v}$. It was first sent by the commander, which is numbered 0. Then, it was received by general $\mathtt{g}_1$, which was received and propagated by general $\mathtt{g}_2$. It was subsequently, received and propagated by general $\mathtt{g}_3$. The entire path is embedded in the message. Note that we can only append a list of signatures to a message. We cannot remove any existing signature. Similar to the majority function, let us define a CHOICE function here. It takes a set of values and returns one of them. It obeys the following properties.

Presence of a default value Assume that $\mathtt{V}$ is empty ($\mathtt{V} = \mathbb{E}$), then CHOICE (V) returns a default value that is known a priori.

Case of a singleton set Assume that $\mathtt{V}$ has only one element $\mathtt{v}$. Then CHOICE ($\mathtt{V}$) is $\mathtt{v}$.

Order independent The output of CHOICE ($\mathtt{V}$) for a vector of values $\mathtt{V}$ is independent of the order of values in $\mathtt{V}$.

Choice criterion The CHOICE function needs to choose one of the values in the set V. It cannot return a value that is not present in V.

Algorithm for the Case with Signed Messages

Let us now look at the algorithm for the case with signed messages (Algorithm 32).

Algorithm 32 Byzantine Consensus with Signed Messages

```
1: procedure INIT
2:     The commander signs an order and sends it to every lieutenant
3: end procedure
4: procedure RECEIVEORDERFROMCOMMANDER(Order v:0, Lieutenant i)
5:     if this is the first order then
6:         Send v:0:i to every lieutenant
7:         V_i ← {v}
8:     end if
9: end procedure
10: procedure RECEIVEMSG(Message v:0:j_1:...:j_k)
11:     i ← GETLIUTENANTID()
12:     if v ∉ V_i then
13:         Add v to V_i
14:         if k < m then                ▷ The algorithm has not terminated
15:             send v:0:j_1:...:j_k:i to all lieutenants other than j_1 ... j_k
16:         end if
17:     end if
18: end procedure
19: procedure TERMINATE
20:     i ← GETLIEUTENANTID()
21:     Wait till either m rounds complete or no more messages need to be sent
22:     Choose CHOICE (V_i) as the consensus value
23: end procedure
```

We start with the INIT procedure. Here, the commander sends an order to every lieutenant. If it is loyal, then only a single copy of its order will be sent. Since the commander's signature cannot be forged, all the lieutenants are bound to receive the same order. Hence, in this case both the conditions for Byzantine consensus hold and it is easy to guarantee the same.

We should concern ourselves with the other case, which is far more tricky – the commander is a *traitor*. In the INIT procedure, the commander sends its order to every lieutenant general. On receipt of the order from the commander, each lieutenant calls the RECEIVEORDERFROMCOMMANDER function. If this is the first order that lieutenant `i` has received from the commander, then it adds the value `v` that is sent to it in a list of values (V_i). Subsequently, it sends the message `v:0:i` to every other lieutenant. This basically means that lieutenant `i` has received the value `v` from the commander – it thus appends its signature. The value that is received is made known to the rest of the lieutenants. Every lieutenant does the same.

Subsequently, the commander exits the picture. The lieutenants run a protocol amongst themselves. They send messages to each other for m rounds, where m is the maximum number of traitors.

When a message is received from a lieutenant, the RECEIVEMSG function is called. We assume that the function GETLIEUTENANTID() returns the id of the lieutenant invoking the function. Any message that is received has the format $v{:}0{:}j_1{:}\ldots{:}j_k{:}i$. This basically means that the value `v` is received from the commander (id 0) and then it is signed by a sequence of lieutenant generals – $j_1 \ldots j_k$. Here $j_1 \ldots j_k$ are the ids of the lieutenant generals. In this sequence there should be no repetition, which basically means that the id of no general should appear twice and all the signatures should be verifiable. A message holding this property is said to be *well-formed*.

Once a message has been verified to be well-formed (not shown in the algorithm), we proceed to check if the value that has been received (`v`) is a part of the set of values that lieutenant `i` has already received (stored in the set V_i). This check is done on Line 12. If it is already there, then nothing needs to be done. However, if it is not there, then there is a need to proceed further.

We add `v` to V_i (Line 13) and check the round number stored in the variable `k`. as mentioned earlier, we run the protocol for a total of m rounds. If $k < m$, which basically means that we are not in the last round, then the value `v` is sent to all the generals other than the generals who have signed the message. The generals who are excluded are $j_1 \ldots j_k$. The main aim of this action is to ensure that whatever a value has been received, it is transmitted to all the generals who perhaps have not seen it (see Line 15). It is thus important to let all such generals know that such a value is in circulation. We would like to reiterate the fact that we are considering the case where the commander is not loyal. Hence, the commander has sent different values to different generals. In the case of this algorithm with signed messages, it is possible to establish that the commander is not loyal because different values

are in circulation with the commander's signature that cannot be forged (if the commander is loyal). Nevertheless, deciding a consensus value is still not easy. There is an inevitable need to send multiple messages between the generals to agree on a consensus value. Note that the final consensus value that needs to be chosen must be one of the values that the traitorous commander had sent.

This process of informing the rest of the generals about a new value that was received by lieutenant `i` in Line 15 has a small twist. Lieutenant `i` appends its signature to the message. Hence, the message looks like this: $\texttt{v:0:j}_1\texttt{:}\ldots\texttt{:j}_k\texttt{:i}$. It is thus clear to the rest of the generals that lieutenant `i` has also seen the value `v`.

Now, note that we are assuming a synchronous algorithm where the beginning and end of a round can be detected. Note that if the message has `k` signatures other than the commander's signature, then we have completed round `k`. Of course, a traitor may decide not to add a signature, but this can easily be detected by a loyal lieutenant and the message can be discarded. It is not well formed in this case. The algorithms proceeds from round to round in this manner.

Let us finally look at the procedure TERMINATE. In the worst case, we wait for m rounds to complete. After that we claim that all loyal lieutenant generals have received the same set of values. This basically means that for generals `i` and `j`, $\texttt{V}_\texttt{i} = \texttt{V}_\texttt{j}$. Then we can simply take the consensus value as CHOICE ($\texttt{V}_\texttt{i}$). As per our claim, this will be the same for all loyal lieutenant generals. Most of the time, this may not be required because prior to this point, the condition in Line 12 that checks if ($\texttt{v} \in \texttt{V}_\texttt{i}$) will evaluate to false for all generals. If all the generals indicate that there is no message to send, then the algorithm can terminate earlier.

A word needs to be added about the feasibility of such algorithms. Everything relies on creating unforgeable signatures. This can easily be achieved with a private and public key based mechanism. Any standard textbook on cryptography [] or secure systems can provide enough background about these techniques.

Proof of the Algorithm

Let us now look at the proof of this algorithm. It is reasonably simple and straightforward. For the case when the commander is loyal, there is no issue. Only one message will be in circulation and thus it will be accepted ultimately by all the loyal generals because for lieutenant `i`, $\texttt{V}_\texttt{i}$ they contain just one element, which the commander had initially sent. The tricky case

is when the commander is a traitor. In this case, the commander can send different values to different lieutenant generals.

Let us assume for the sake of argument that there is a value $\mathtt{v}'$ such that $\mathtt{v}' \in \mathtt{V_i}$ and $\mathtt{v}' \notin \mathtt{V_j}$ for two loyal lieutenants $\mathtt{i}$ and $\mathtt{j}$. Let us consider the moment in which value $\mathtt{v}'$ was added to $\mathtt{V_i}$. First, assume that it was a value that was sent by the commander. Then in the very next round, this value would have been sent to the rest of the lieutenants that would include $\mathtt{j}$. This means that $\mathtt{j}$ would have $\mathtt{v}'\mathtt{inV_j}$. However, this is not the case. This means that $\mathtt{v}'$ was sent in a later round to lieutenant $\mathtt{i}$.

Assume the message was $\mathtt{v}'\mathtt{:0:j_1:\ldots:j_k}$. Note that lieutenant $\mathtt{j}$ cannot be in the list $\mathtt{j_1 \ldots j_k}$, otherwise it would have already seen $\mathtt{v}'$. If this is not the case, then lieutenant $\mathtt{i}$ would send the value to $\mathtt{j}$ as per the logic in Line 15 assuming that the round number is not equal to m. $\mathtt{j}$ would then get $\mathtt{v}'$ and add it to $\mathtt{V_j}$ if $\mathtt{v}'$ is not already there. Hence, this case is not possible.

The only case that is remaining when $\mathtt{k} = \mathtt{m}$ and $\mathtt{i}$ receives $\mathtt{v}'$ in the last (m^{th}) round. Now, consider the list of lieutenants $\mathtt{j_1 \ldots j_m}$. Given that the commander is a traitor, this list can have at the most $m-1$ traitors. This means that at least one of the generals in this list is loyal. Let this general be $\mathtt{j_x}$. $\mathtt{j_x}$ will send the value $\mathtt{v}'$ to lieutenant $\mathtt{j}$ (see Line 15). Hence, $\mathtt{j}$ will learn the value.

Our analysis indicates that there is no case in which $\mathtt{j}$ will miss the value $\mathtt{v}'$. Hence, for any two loyal lieutenant generals, $\mathtt{i}$ and $\mathtt{j}$, $\mathtt{V_i} = \mathtt{V_j}$. As a result CHOICE ($\mathtt{V_i}$) or CHOICE ($\mathtt{V_j}$) will have a fixed value for all loyal lieutenants.

This satisfies the requirements of Byzantine consensus.

Message Complexity

There is sadly no appreciable decrease in the message complexity when we have a traitorous commander – it is still exponential in terms of the number of traitors. The exact expressions are complex and non-intuitive. Let us provide a very simple reasoning. Consider the sequence of signatures in any round. They will be of the form – $\mathtt{v:0:j_1:\ldots:j_k}$ – in the k^{th} round. Note that if one of these signatures is by a loyal general, then it means that it will broadcast the value to all the generals who haven't signed and by the beginning of the next round, all the loyal generals will get the value. Traitorous generals will not be able to create new copies of this message with altered values because a loyal general has already signed (its signature cannot be forged). Hence, they will not see any value in a message that has been signed by a loyal general because they are sure that all loyal generals

have received a copy of the value.

However, traitorous generals can unfortunately create new messages from thin air that bear the signatures of other traitorous generals only. We can think of traitorous generals colluding and having access to each other's signatures. They basically can forge each other's signatures with full consent. As a result, it is possible to generate $m(m-1)\dots(m-k+1)$ messages that bear k signatures of traitorous generals. All of these messages can have different values and can be sent to the rest of the $n-m$ loyal generals. For every round, the traitorous nodes can general such messages that have all the combinations and permutations of signatures of traitorous generals. The associated values also can all be different. This will ensure that the loyal generals do not discard the messages.

This process can continue right up till m rounds.

As a result, the number of messages is in the ballpark of $m!$ (set $k = m$), which is basically exponential in terms of the number of traitors, m.

6.4 The Paxos Algorithm

We know that we are naturally constrained by the FLP result when it comes to designing a consensus protocol in an asynchronous setting with at least one faulty process. However, it is important to understand that an asynchronous setting with one or more faulty processes mimics most real-world scenarios. Consider the case when we are trying to book an airline ticket. We clearly need a consensus between the app running on the phone, the airline, the travel agency, the credit card company and the bank that issued the card. Only if all of them agree, then only the transaction can go through and a ticket can be booked. Otherwise, even if one of them disagrees, then the transaction needs to be aborted. Here again, there needs to be complete agreement about aborting the transaction. In other words a consensus needs to be reached.

Many times it so happens that money has been deducted from a bank account but the ticket has not been booked. This is a sad reality of a lot of travel sites that do not implement consensus protocols properly. This causes a lot of headache for passengers and then they find themselves requesting the travel site on the phone to either give them a ticket or refund the money. It is thus important that we design consensus protocols for such settings, which at least work most of the time.

The FLP result basically says that we cannot guarantee both *safety* and *liveness*. In this case, safety basically means that the processes do not end up

deciding different values. Liveness means that for every process, a consensus decision is made within a finite or preferably bounded amount of time. Because of the FLP result, it is clear that we need to sacrifice either safety or liveness. Clearly safety cannot be compromised, hence liveness needs to be the casualty here. This means that in a rarity of cases, the protocol will not terminate. We will just keep on trying to achieve a consensus over and over again, yet a consensus will never be reached. This scenario is also known as a *livelock* where every process continues to take internal steps but is never successful.

Definition 47 *A* livelock *is defined as a situation in a concurrent system where every process continues to make progress by taking internal steps, however no process is ever successful in achieving its desired outcome, i.e., a consensus. Despite the seemingly constant progress, it is possible that for indefinite time no process will ever reach the consensus state (finally decide on a value). This is not the same as a deadlock, where processes are not able to make any progress at all and there is a cyclic dependency between them.*

6.4.1 Basic Concepts

Let us now proceed to discuss the basic concepts underlying the classical Paxos algorithm that was originally proposed by Leslie Lamport [77]. The algorithm was fairly complex and a lot of people did not understand it. As a result, its adoption was initially quite difficult. Later authors including Leslie Lamport himself published simplified versions of the Paxos algorithm that were far more straightforward and easier to understand. The version of the algorithm that we will be presenting in this chapter is a much more simplified version of Paxos; it was published in 2001 and is referred to as simplified Paxos [79].

Here, the overall idea is quite simple. We have three kinds of processes in the system: a proposer, an acceptor and a learner. The job of the proposer is to propose values that will be considered by the consensus protocol. The job of an acceptor is to temporarily accept values and ensure that the protocol reaches termination. A learner joins the system much later and tries to learn the value that was ultimately chosen as the consensus value.

6.4.2 Conditions

Let us define a set of conditions that must hold at all points during the operation of the algorithm. They may look like being overly conservative,

however they are needed to realize a fast, simple and efficient consensus protocol in an environment that is asynchronous as well as contains slow/faulty processes.

C1: First-Accept Condition

An acceptor does not know how many proposals are currently there in the system. She has a limited view of the system. Hence, she needs to *accept* the first proposal that she gets

Let us understand the rationale behind C1. Many values can be simultaneously proposed by different proposers. Other processes will not be aware of all of them. This is because in a concurrent system a process has a very limited view of the system. It will only aware of its own state and has a sketchy view of the global state based on the messages that it has received. This means that an acceptor process needs to possibly accept multiple proposals. In this case, proposal acceptance is a temporary step. This is not the same as finally *choosing* or deciding on a proposal. It is a provisional acceptance.

Now, consider the first proposal that an acceptor gets. She does not know if this will be the last message or not. It very well may be the last proposal that she gets. Hence, she has no choice but to accept it.

Before introducing the next condition, let us outline the structure of a proposal. It is a tuple of number and value.

$$proposal \rightarrow (number, value)$$

The proposal numbers can be lexicographically ordered – this makes all the proposal numbers unique. Let us now come to the second condition, which is known as the Consensus condition (C2).

C2: Consensus Condition

If a proposal (n, v) is *chosen*, then every proposal that is chosen with a higher number, needs to choose the value v.

C2 basically establishes consensus. It essentially establishes the finality of consensus. Once a value is chosen, then no other value can be chosen. We can look at this another way. Take the least proposal number that is chosen. Note that all the proposal numbers are comparable here, hence the notion of the "least number" exists. Now, every higher-numbered proposal needs to choose the same value as per the definition of consensus.

To satisfy C2, we need to define two sub-conditions, which are actually more restrictive.

C2a: Acceptance Condition

If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor also has value v.

Condition C2a makes things more restrictive. It basically puts a restriction on proposal acceptance as well. It says that after a proposal is *chosen* by the system, no proposal that conflicts with the chosen value can be accepted. This means that if we have chosen v, henceforth, we can only accept v. No other value can be accepted.

Now, this seems to violate condition C1, which is the First-Accept condition. Assume a process was sleeping. It suddenly wakes up. By the First-Accept condition it needs to accept the first proposal that it gets. This could very well be a proposal that conflicts with the chosen value. Given that the process that just woke up does not have any additional visibility, it needs to accept the proposal. Even though this satisfies condition C1, it violates condition C2a.

We thus need to add one more condition to ensure that this does not happen. This will basically make things more restrictive.

C2b: Issuance Condition

If a proposal with value v has been chosen, then every higher-numbered proposal issued by any proposer can only have value v.

In this case, we are placing a restriction on issuing proposals. We are saying that no process can even issue proposals with other values once a consensus decision has been made. This means that the sleeping process in the counter-example that we just discussed for condition C2a will never get a proposal that has any value other than v. Hence, it can happily use the First-Accept condition (C1) to accept the first proposal that comes to it. This will not cause a problem.

Basically, C1 is a necessity because every process needs to make forward progress – in a certain sense it ensures liveness. However, for consensus, we need to ensure condition C2b.

6.4.3 The Algorithm

The algorithm is very short and simple. It is quite surprising that it works. The algorithm has two phases. Phase 1 (run by the proposer) is shown in Algorithm 33. It starts with a call to the method PROPOSE.

Every process has an internal variable called `cnt` (counter) and `val` (value). The count is incremented when the PROPOSE method is invoked with argument `v`. First we initialize the process variable `val` to `v`. `val` is the value that the given process is going to propose if it is allowed to do so.

There are two more internal variables: `maxPrep` and `maxAccept`. `maxPrep` is the number of the highest-numbered proposal received by the current process. `maxAccept` is a proposal, which is a (number,value) tuple. It is the highest-numbered proposal accepted by the current process.

Algorithm 33 Paxos: Phase 1

```
1:  procedure PROPOSE(v)
2:      ▷ Called by the proposer
3:      cnt ++
4:      val ← v
5:      Send ⟨prepare(cnt)⟩ to a majority of acceptors
6:  end procedure
7:
8:  procedure RECEIVEPREPARE(n)
9:      ▷ Called by the acceptor
10:     if n > maxPrep then
11:         maxPrep ← n
12:         return maxAccept
13:     end if
14: end procedure
```

Whenever a process wants to propose a new value, it increments `cnt` and sends a prepare message with `cnt` to a majority of acceptor processes (referred to as the *quorum* henceforth). The reason that we use a *majority* here is because if another process does the same, then there will at least be one acceptor in common. We will see that the common acceptor plays an important role in the proof.

When a process receives a ⟨`prepare(n)`⟩ message, it calls the RECEIVEPREPARE function. This is where it compares the number of the proposal with the internal variable `maxPrep`, which is the highest proposal number that it has received. It wishes to consider only higher-numbered proposals. It

checks if `n` > `maxPrep` in Line 10. It enters the body of the *if* statement only if `n` is greater. This means that it always wishes to consider newer proposals. In this case, it sets `maxPrep` to `n` and returns a message with the value of the internal variable `maxAccept`, which is actually a proposal – a tuple of the proposal number and proposal value. We will henceforth refer to them as `maxAccept.n` and `maxAccept.v`, respectively. Note that here `maxPrep` is a monotonically increasing variable. `maxAccept` is the highest-numbered proposal that the current process has accepted. Note that if no proposal has been accepted as yet, then a null value needs to be returned.

There is a fine print here that is not easily visible. The first point to note is that if `n` < `maxPrep` then the message is pretty much absorbed, nothing is returned. However, if `maxAccept` is returned, then it is returned along with a *promise*. The promise is that no proposal with a number less than or equal to `maxPrep` will ever be entertained in the future. This promise is important because it helps us clean up lower-numbered proposals from the system and ensures the monotonicity of `maxPrep`, which is important for guaranteeing consensus.

Once the messages with `maxAccept` proposals are received from all the acceptors, the algorithm proceeds to the second stage (shown in Algorithm 34). Note that if a process sends a proposal with a low number, it will simply not get replies from many acceptors because the check `n` > `maxPrep` will fail. We shall proceed to Phase 2 only if a majority of acceptors reply with their `maxAccept` proposals.

Summary of Phase 1

Phase 1 of the algorithm achieves two things for the proposer: First, it extracts a promise from each acceptor that it will never respond to a proposal with a number less than the proposal number n. Second, it gets the highest-numbered proposal that accepted by each acceptor (null if no proposal has been accepted).

When a process receives replies from a majority of acceptors, it transitions to Phase 2. This informally means that the number of its proposal is high, which is why it got all the replies. It also means that when each of the acceptors had gotten the current proposal in Phase 1, they had not replied to any higher-numbered proposal. Hence, entering Phase 2 indicates the recency of the proposal. It also indicates that each of the responding processes will not accept a lower-numbered proposal.

Algorithm 34 Paxos: Phase 2

```
1: procedure receiveMaxAccepts(maxAccepts[])
2:     ▷ Called by the proposer
3:     ▷ received maxAccept messages from a majority of acceptors (quorum)
4:     (n, v) ← max(maxAccepts)
5:     if (n, v) = φ then
6:         (n, v) ← cnt, val
7:     else
8:         (cnt, val) ← (n, v)
9:     end if
10:    send ⟨accept(n, v)⟩ to all the acceptors in the quorum
11: end procedure
12:
13: procedure receiveAccept(n, v)
14:    ▷ Called by the acceptor
15:    if n ≥ maxPrep then
16:        maxAccept ← (n, v)
17:        accept the proposal (n, v)
18:        send a ⟨response⟩ to the proposer
19:    end if
20: end procedure
21:
22: procedure receivedAllResponses
23:    ▷ Called by the proposer
24:    choose val
25: end procedure
```

In the function receiveMaxAccepts, we first find the maximum numbered proposal out of all of the `maxAccept` proposals received (from the majority of acceptors). We put all the `maxAccept` proposals received in the array `maxAccepts` and then find the maximum. Let us refer to this proposal as (n, v) (number,value). If all of the received proposals are null because no proposal has been accepted till now, then the proposer uses the value that it is proposing (cnt, val). Note that this is the only instance, when the proposer gets to propose its own value. Condition C2b says that once a decision has been made we need to stop proposals with other values even from getting issued (sent to the other processes). This is the point at which this is enforced. A process only gets to *propose* if no other process in its

quorum has accepted any proposal. The next step is to send $\langle \texttt{accept}(\texttt{n}, \texttt{v}) \rangle$ to all the acceptors in the quorum.

When an acceptor gets this $\langle \texttt{accept} \rangle$ message, it invokes the function receiveAccept. The arguments are the proposal number `n` and the proposal value `v`. The acceptor compares `n` with `maxPrep` again (see Line 15). Recall that in Phase 1, this comparison was made for the first time (Line 10 in Algorithm 33). The reason for comparing for a second time will be clear in the proof. However, informally it means that no other proposal was received in the intervening period with a higher number.

There is a general point to be made here. Many distributed algorithms work on the principle of the "period of quiescence". This means that they rely on a small period of time when no other message has been sent or received – a period of quietness or *quiescence*. The existence of such a period can of course be checked with checking a variable twice as we are doing in Line 10 in Algorithm 33 and Line 15 in Algorithm 34. We shall see that having such a period of quiescence has one advantage and one disadvantage. The advantage is that we can prove that in the distributed system some global state must have been reached – all or majority of the nodes must have agreed on some property. However, the disadvantage is that we may never have such a period of quiescence and the algorithm will just continue forever. This, in fact, is a negative side of consensus algorithms that guarantees safety but not liveness. Recall that this is a direct consequence of the FLP result.

Now, let us restart our discussion from Line 15 (Algorithm 34). If the $\texttt{n} \geq \texttt{maxPrep}$ check succeeds, then we set `maxAccept` (internal variable of the process) to the received proposal $(\texttt{n}, \texttt{v})$. This is tantamount to **accepting** the proposal $(\texttt{n}, \texttt{v})$. A response can be sent to the proposer informing it about the same.

Once all the responses are received, the proposer can end Phase 2 by invoking the function receivedAllResponses. The value `val` (value that was proposed in the $\langle \texttt{accept} \rangle$ message) can now be chosen as the consensus value.

We need to appreciate how the First-Acceptance condition (C1) and the Proposal Issuance condition (C2b) are being ensured. Consider the first message that an acceptor gets. It is simply accepted unless another higher-numbered proposal has contacted it between Phase 1 and Phase 2. If there is a period of quiescence where no such higher-numbered proposal is received, then the first proposal is accepted. A subsequent proposal can also be accepted by an acceptor as long as it satisfies the condition $\texttt{n} \geq \texttt{maxPrep}$.

Once a value is chosen, it is clear that a majority quorum has set its

`maxAccept` value. This is because at the end of Phase 2, the ⟨`accept`⟩ message (with the value that will be used to set `maxAccept`) is broadcast to a majority of the processes and we wait for their response. Once the responses are received, we can be sure that they have set their respective `maxAccept` values. Now, the claim is that no new proposal can be issued or in other words enter the system after this (C2b). This is clearly going to happen because in the beginning of Phase 2, the proposer asks the majority quorum for their `maxAccept` values. After a proposal is chosen, this set will have at least one non-null value. Thus the condition for adding a new proposal to the system is not getting satisfied here. A new proposal can only be added only if all the `maxAccept` values received are null, which will not be the case here.

Hence, we see that this algorithm is obeying both the conditions: C1 and C2b.

Summary of Phase 2

The proposer receives a set of `maxAccept` proposals from the majority quorum. It finds the highest-numbered proposal and if all are null, then it makes a proposal. This is then multicasted back to the majority quorum. If no higher-numbered proposal has arrived in the time being, then an acceptor **accepts** the value that is being sent to it by the proposer. This involves setting the value of `maxAccept` and sending a response back. Once the majority quorum sends the response back, the value sent in the ⟨`accept`⟩ message can be chosen.

6.4.4 Analysis of the Paxos Algorithm

Consider two proposals P_1 and P_2. Consider an acceptor A that receives P_2's ⟨`prepare`⟩ message after P_1's ⟨`accept`⟩ message. Then we say at A, P_1 precedes P_2 or alternatively $P_1 \prec P_2$. This means that pretty much in this two-proposal setting, the ⟨`prepare`⟩ message is received after the period of quiescence at A.

Now, let's say P_1's ⟨`prepare`⟩ message is ignored (does not pass the check in the *if* statements). Assume that A receives P_2's ⟨`prepare`⟩ message after ignoring P_1's ⟨`prepare`⟩ message. Then also we can say that at A $P_1 \prec P_2$.

It is possible that both the ⟨`prepare`⟩ messages are *concurrent*. This means that neither $P_1 \prec P_2$ nor $P_2 \prec P_1$ holds true. Then we say that $P_1 \bowtie P_2$. This is equivalent $P_1 \not\prec P_2$ and $P_2 \not\prec P_1$ (at A). We say that $P_1 \bowtie P_2$, if $P_1 \bowtie P_2$ at any acceptor.

Let us now prove a bunch of lemmas regarding the properties of the protocol.

Lemma 48 *If $P_1 \bowtie P_2$ and $P_1.n < P_2.n$, P_1 will not pass Phase 2 at the common acceptor.*

Proof: Given that these messages are sent to a majority quorum, there must be a common acceptor. This is because an intersection of two majority sets is always non-empty. Let us thus assume that there is a common acceptor A. Assume that it got the prepare message first from P_1. Given that $P_1 \bowtie P_2$, P_2's message will arrive before P_1 crosses Phase 2. It will thus either set the value of `maxPrep` or `maxPrep` will be greater than $P_2.n$. In either case `maxPrep` will be greater than $P_1.n$, and thus P_1 will fail the check in Line 15 (Algorithm 34) and it will not pass Phase 2.

Now assume the other case where it first gets the $\langle$`prepare`$\rangle$ message from P_2. In this case, `maxPrep` $\geq P_2.n$. As a result, P_1 will fail the check where `n` is compared with `maxPrep`. It will thus fail Phase 1. Hence, the question of passing Phase 2 does not arise. □

Lemma 49 *If $P_1 \prec P_2$ and $P_1.n > P2.n$, P_2 will not pass Phase 1.*

Proof: Let us again consider the common acceptor A. P_1 would have either set the value of `maxPrep` in Phase 1 or found out that `maxPrep` is greater than its proposal number. This means that when P_2 arrives, it will definitely `maxPrep` to be greater than its number in Phase 1. It will thus fail Phase 1. □

Let us understand what the lemmas are giving us in terms of results. Lemma 48 says that it is not possible for two proposals to pass both the phases at all acceptors if they are concurrent. Lemma 49 further refines this notion. It says that the proposal numbers should obey the same relationships as the precedence relation. If $P_1 \prec P_2$, then $P_1.n < P_2.n$. This basically means that if two proposals P_1 and P_2 succeed (pass Phase 2 at all acceptors) then with no loss of generality $P_1 \prec P_2$ and $P_1.n < P_2.n$. We now want to prove the main theorem, which is that for all such pairs of proposals, they end up *choosing* the same value.

Theorem 50 *If $P_1 \prec P_2$ and both of them succeed in passing Phase 2 at all acceptors, then they must choose the same value.*

Proof: We know that $P_1.n < P_2.n$ (from Lemma 48 and Lemma 49).

Assume that they choose different values. P_1 chooses v_1 and P_2 chooses v_2. P_1 and P_2 must have a common acceptor A. After A received P_1's

⟨`accept`⟩ message, it must have received P_2's ⟨`prepare`⟩ message. It must have sent it the proposal containing value v_1.

However, for some reason P_2 did not choose it. This is basically because it must have a gotten a higher-numbered `maxAccept` proposal with value v_2 from another process in its quorum. Let the proposer of v_2 be P_3.

The question is, how did P_1 miss v_2 (proposed in proposal P_3)? Since P_3's proposal could beat P_1's proposal, it means that $P_3.n > P_1.n$. Given that P_1 succeeded, it means $P_1 \prec P_3$. P_3 must have solicited `maxAccept` messages from a majority quorum. It would have clearly seen a non-null value because by this time P_1's accept message would have reached a majority quorum. This means that the proposer of P_3 could not have proposed a new value, i.e., P_3's value. This is because a new value can only be proposed if all the `maxAccept` messages are null, which in this case is not going to happen. As a result, P_3's value cannot be issued as a proposal (in line with Condition C2b).

Since P_3's value was never circulated, it could not have been accepted by P_2. There is thus a contradiction here.

Hence, P_1 and P_2 will choose the same value. In fact, once a value is chosen, no other value can be chosen. □

6.5 The Raft Consensus Protocol

The Paxos protocol that we just described is basically for a single consensus round. If we want multiple rounds, then it starts becoming more complex and starts approaching the original Paxos protocol that Leslie Lamport had proposed []. Also, the Paxos protocol is a classical concurrent algorithm where a majority of nodes need to periodically participate in the consensus process. To make a decision, a period of quiescence is required. The community found this class of algorithms to be very difficult and thus the acceptability of such complex protocols has always been quite low. Hence, for a long time, people wanted to design a much simpler protocol, which is very easy to implement, understand and verify. We need to bear in mind that if a protocol is complex, then coding and verifying it also becomes a very onerous task. Furthermore, its code does not remain maintainable because most developers find the algorithm and its code very hard to understand. Keeping all of these requirements in mind, Raft [93] was proposed in 2014.

The Raft protocol operates like a distributed ledger (similar to many cryptocurrencies). There are many clients and servers. One process is elected as a *leader process*. Each server maintains a state machine. The

state machines across the servers are expected to be synchronized. Client machines send their requests (proposals in Raft) to the leader server. A request performs some operation on the state machine and changes its state. All the servers need to agree on the global order of the requests. If they do (consensus condition), then they can compute the current *state* by applying all the requests one after the other to each state machine. Given that all the servers already agree on the order of requests, the final state will be the same in each state machine, even though all the servers compute it separately. This is a standard paradigm in distributed systems. The current state is a result of a sequence of transitions applied to a state machine that is initialized to a given state.

Let us outline the role of a *leader*. It accepts requests from clients and multicasts them to the servers. Furthermore, it establishes a global order of client requests and conveys this global order to rest of the servers. The servers then know the order in which these requests need to be processed (applied to the state machine). This will ensure that all the state machines remain *synchronized*.

There is a possibility that a leader fails. In this case, a new leader needs to be elected. The new leader may fail again. It is also possible that a leader that was hitherto known to have failed suddenly wakes up. There is thus a need to divide time into *terms*, where each term has a unique leader. To ensure that an old leader that had gone to sleep does not mistakenly assume that it is still a leader, we associate a monotonically increasing sequence number with each term. This helps us identify leaders from previous terms. It is possible to make them realize that they are not leaders anymore.

Before we proceed further, a disclaimer is due. The Raft protocol works in spite of node failures, network packet losses and network partitions. However, it is not resilient to Byzantine failures. This means that it assumes that processes can fail and be slow, however they do not lie.

6.5.1 Safety Properties

Property	**Explanation**
Election safety	At most one leader can be elected at a given point in time
Leader append-only	A leader never overwrites or deletes an entry in its log. New entries are only appended.

Log matching	Consider two logs that contain the same entry at a given index and term. The logs are identical till that index.
Leader completeness	If an entry is *committed* in a given term, it will be present in the logs of the leaders of all successive terms.
State machine safety	There is a consensus on logs when they are applied to a state machine. Furthermore, if a log entry at a certain index is *applied* to a state machine, then all the servers will also apply it to their respective state machines.

Table 6.2: Safety Properties in Raft

Let us understand each of these safety properties. We divide time into a set of *terms*, where each term can have at most one leader. This is known as *election safety*. In a certain sense, we are creating a centralized algorithm. However, we are accounting for the fact that a node may fail and thus it would be necessary to elect a new leader. In this case, we create a new term and elect a new leader. The log is basically a linked list that comprises all the changes that need to be applied to the shared state machine. This is immutable, which basically means that it is not possible for a leader to overwrite or delete entries in the log – it can only *append* new entries.

There is a need to reconcile the logs across servers especially when there are leader changes. Hence, for each entry we also add an index, which is pretty much its sequence number in the log (starting from 0). In essence, we are treating a log as an array where the index can be interpreted as the array index. The *log matching* condition states that if two logs contain the same entry at a given index, then the logs are identical till that index. This can be thought of as a consensus condition on the logs. We shall see later that the algorithm guarantees a far stronger condition, which says that if two servers have j and k entries respectively in their logs where $j < k$, then the larger log needs to contain all the entries in the smaller log.

Raft distinguishes between propagating a state machine update to all servers and committing it. Whenever a client makes a request, it is sent to the leader server. The leader adds it to its log in an uncommitted state and then broadcasts it to all the servers. Only after getting a response from a majority of servers does it actually *commit* it. In this case, commitment means that the entry is permanently added to the log. Logs thus have a

certain sense of immutability. Raft guarantees that if an entry is *committed* in a given term, it will be present in the logs of the leaders of all successive terms. This is precisely implied by the notion of commitment. This property is known as *Leader Completeness*.

The final condition *state machine safety* is the consensus condition. It says that if a server has *applied* a log entry to the state machine – made it transition according to the entry – then all the other servers will also apply the same entry (stored at the same index) to the state machine. All the servers need to basically see a common view of the log. Let us elaborate.

Periodically, the leader server sends messages to the rest of the servers. It ensures that all of its committed entries reflect as committed entries in other servers as well. If an entry is committed in a log, then we assume that all of its previous entries (with lower indexes) are committed as well. It further means that all the servers will eventually see the same order of committed messages in their logs. A log can then be *applied* to a state machine. Applying a log means that we traverse the log from the beginning till the last committed entry and make the state machine make transitions as per the contents of the log entry. Note that we can apply a log entry only if it is committed.

6.5.2 Overview

Figure 6.8 shows an overview of the scheme. A Raft cluster typically contains 5 or more servers. A server has three states: leader, follower and candidate. The names are self-explanatory.

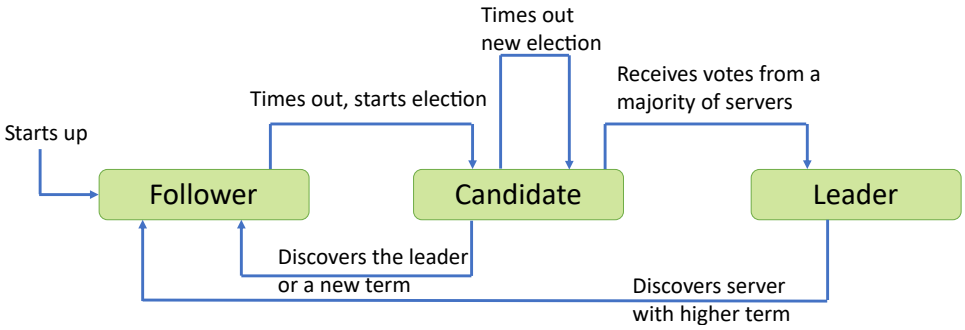


Figure 6.8: Raft state diagram

Let us now take a deeper look at Figure 6.8 and explain all the state transitions. The system starts in the follower state and then leader election

starts. This is when an interested follower becomes a candidate. If a candidate discovers a new leader, then it realizes that the election has already happened. It thus relegates itself and becomes a follower. On the other hand, if there is a timeout and no leader is elected, then a new election is started. Once a candidate gets the majority vote, it becomes a leader and remains a leader until the next term. From the leader state, the only transition that is possible is to the follower state. This happens when another server is discovered that has a higher term. This server could also be the new leader that has replaced the current leader unbeknownst to it.

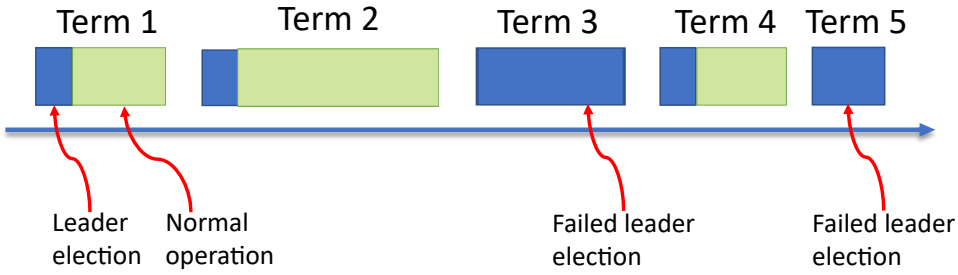


Figure 6.9: Division of time into terms

Figure 6.9 shows how we divide time into *terms*. As we can see, every term is numbered. These numbers are maintained by each server. A term begins with the leader election, then there is normal operation of the protocol where log entries are added to the log. Finally, when a term ends, a new term begins. It is possible that the leader election process fails and no leader is elected. In this case, normal execution cannot happen in that term.

The term number needs to be attached to every message. This is akin to timestamping every message. Using this method we can find messages that belong to old terms or messages that have simply been delayed. If a server with a higher term number receives a message that is stamped with a lower term number, then it drops the message. This should be done because we do not want any server to take cognizance of messages that were sent in previous terms. These are old and outdated messages and thus should not be processed. Assume we have the reverse case – the server receives a message with a higher term number. In this case, it needs to upgrade its term number and set it equal to the higher term number. This is very similar to Lamport clocks.

The term number also plays an important role in the election process. If a candidate or a leader process finds that its term is *stale*, which basically

means that it has received messages with higher term numbers, then it relegates itself and becomes a follower once again. We basically want the system to remain up to date at all points of time: this includes normal operation as well as the leader election process.

6.5.3 Leader Election

All the servers start in the follower state. They periodically get ⟨`heartbeat`⟩ messages from the leader that also contains its term. This indicates that the leader is alive and there is no network partition. If a server does not get a ⟨`heartbeat`⟩ message for a certain duration, then it is clear that something is wrong. It times out and begins the process of leader election.

The process of beginning an election is quite simple. The server increments the current term number, transitions to the candidate state, votes for itself and sends a ⟨`RequestVote`⟩ message to the rest of the servers. Note that a server can only vote for one candidate in a term. This is similar to a regular election. Any server process will obviously vote for the candidate that sent it the first ⟨`RequestVote`⟩ message because it does not know how long it needs to wait if it wants to vote for the second ⟨`RequestVote`⟩ message that it receives (if at all). Recall that we had something similar in Paxos. We called it the *First-Accept* condition.

Now, there are three possible outcomes of the election process.

Wins the election: In this case, a server gets the majority of the vote. It clearly knows that it is the leader. It thus goes on to declare itself as the leader and starts sending ⟨`heartbeat`⟩ messages to the rest of the servers. Since we are not assuming Byzantine faults, it is not possible for two servers to declare themselves as leaders in the same term.

Receives an ⟨`AppendEntries`⟩ message from a server: This is a regular message to append entries into the log. If the term in the message is greater than the current term, then it is necessary to transition to the follower state and recognize the server that sent the message as the leader. We shall see later that these ⟨`AppendEntries`⟩ messages are sent only by the leader. No other server can send such messages.

No leader is elected This means that a situation was reached when the server did not get a majority of the votes. Such a situation is known as a *split vote*. There is no option but to attempt the leader election once again with an incremented term number. The backout or quiescence

period can be randomized to ensure that conflicts are reduced and at least one candidate has a better chance of winning the election.

6.5.4 Managing the Logs

After a leader has been elected, it makes its presence known to the clients and other servers by sending messages. The entire system knows who is the current leader unless there are network partitions. Clients send requests to the leader to append entries to the log. The leader then sends entries to the rest of the servers using ⟨`AppendEntries`⟩ messages. A server can either drop the message, request for reconciliation of logs or acknowledge the message by sending a response.

Let us now look at the structure of a log. It is a list of entries, where each entry stores a term number and a command. This command is meant to be *applied* to the state machine, whose updates are supposed to be synchronized across the entire system. Furthermore, each entry has an index that indicates its position in the log. It is like an array index – a monotonically increasing integer.

An entry is considered *final* or an unremovable part of the log when it is *committed*. An entry is said to be committed when it has been sent to a majority of the servers and the same has been accepted by them. Committing one entry is tantamount to committing all the preceding entries as well (implications of this discussed later). This is enforced by attaching the highest committed index along with every message. This tells each server about all the indexes that can be committed.

Log Matching

Let us now look at the Log Matching property in some detail. We wish to state two sub-properties.

S1 If two entries in different logs have the same index and term, then they store the same command.

S2 If S1 holds for a given index, then all the preceding entries of the logs are *identical*.

The Log Matching property is ensured as follows (proven later). After the leader creates an entry at a given index and term, it broadcasts it to its followers. Now, to ensure S1 and S2, the leader also appends the index and term of the latest entry in its log (before adding the current entry) with the

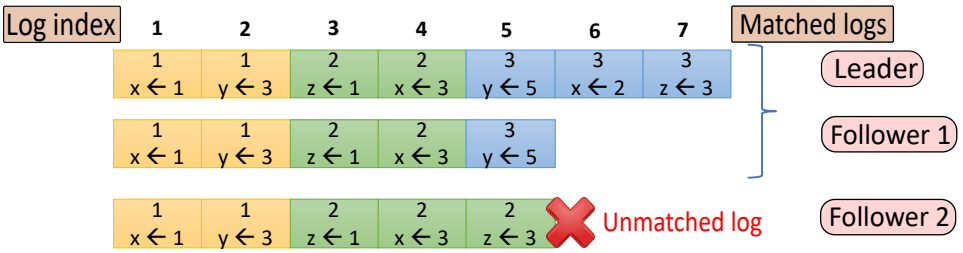


Figure 6.10

⟨`AppendEntries`⟩ message. The follower needs to check the latest entry in its log. If its term and index match the index and term sent by the leader, then it means that its logs are in sync with the leader. As per the Log Matching property, we expect the rest of the entries to also be identical.

This is shown in Figure 6.10. The logs of the leader and Follower 1 match. However, for the third log at the bottom corresponding to Follower 2, we see an unmatched log entry.

Reconciling Log Entries

Let us now consider the case when there is a mismatch in the index and term. The leader needs to fix the entries in each follower to match its own because the leader never deletes or overwrites entries. To do this, the leader maintains a *nextIndex* pointer for each follower (see Figure 6.11). This pointer is initialized to one plus the index of the last entry in the leader's log. This means by default it assumes that all the entries match unless proven otherwise.

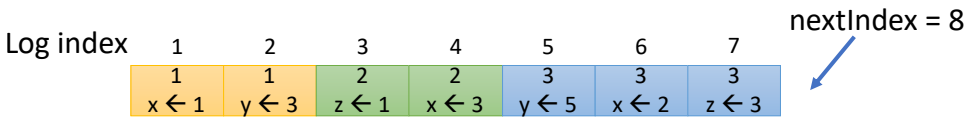


Figure 6.11: A log with a *nextIndex* pointer

For some followers there may be a log mismatch. This can be detected by simply comparing the latest entries of the logs of the leader (prior to adding the new entry) and a follower as discussed earlier. Note that we are throughout assuming the Log Matching property, which says that if two entries at the same index match, then all the entries in earlier (smaller)

indexes also match. This will be proven later on.

Now, after receiving a message from the leader, a follow may find that the respective entries at $nextIndex - 1$ do not match. This means that the logs are not consistent up till that point. We need to find the latest point (greatest index) at which the logs begin to diverge. The leader continues to decrement *nextPointer* for the follower and continues to try. Ultimately, a point will arise when the logs match. Till this point, both the properties S1 and S2 are satisfied – not beyond it. We thus can conclude that all the entries in the follower's log beyond the point of match are invalid. They are thus removed.

The leader can then send the remaining log entries that succeed the point of match to the follower. The follower needs to simply add these entries to its log. It will basically be replicating the leader's log by doing this.

It is important to note that it is the follower that is forced to delete/overwrite entries in its log. The leader never overwrites/deletes entries; it only appends.

6.5.5 Dealing with Server Crashes

Leaders will continuously keep changing in this protocol. However, the new leader is bound by the same commandments as the previous leader – it cannot delete or overwrite any entries. Furthermore, they need to preserve all the committed entries that have possibly been committed by previous leaders.

The Leader Completeness property is enforced as follows. For a candidate to win an election, it needs to ensure that it contains at least all the committed entries (Leader Completeness). The ⟨`RequestVote`⟩ message includes information about the candidate's log and has a count of the number of committed entries. Needless to say it should be at least as up to date as the log of each voter. The check to find if a candidate's log is up to date or not is quite simple. There are two conditions.

- Check the last entries of the logs. The entry with the higher term is more up to date.
- If the terms are the same for the last entries, the log with more entries is more up to date.

Now, assume that a leader crashes. Consider an entry e. If the leader server crashes before *committing* e in a majority of servers, then we have a complex situation in our hands. The log of the new leader has to be as up to

date as a majority of the servers (the voters). Assume that the new leader has e in its log, then there is no problem. It will ultimately commit it and send it to all the servers. However, if it does not have the entry, then the new leader has a choice. It can either proceed to commit or ignore it. It is a much better idea if it chooses to ignore it – after all it is an uncommitted entry from a previous term.

Let us now consider the case when a follower or candidate crashes. Raft continuously tries to send ⟨`AppendEntries`⟩ and ⟨`RequestVote`⟩ messages to all the servers. This includes followers and candidates that may have crashed. Given that all these messages are idempotent (can be sent multiple times), there is no problem if these messages are sent many times in case the feeling is that they are getting lost.

There is however a timing requirement for the protocol to execute reasonably well. The time need to broadcast a message should be much lower than the election timeout time, which should ideally be much lower than the mean time between failures.

6.5.6 Proof

Given that we have described the algorithm, we need to now prove that the five properties mentioned in Section **??** hold.

Election Safety

In any term, we can only elect a single leader or we will have a situation with split votes (no majority). This is because in a given term, a server can vote only once, and we need a simple majority to elect a leader. Hence, two leaders cannot be elected.

Leader Append-Only

By design, the leader does not append or overwrite any entries in its log. This is ensured by design.

Log Matching

Let us prove that the described algorithm maintains the Log Matching property. We will prove a series of lemmas to reach our final goal.

Lemma 51 *If server A has N_A entries in its log for term T and server B has N_B entries in its log for the same term T, and $N_A \geq N_B$, then A contains all the entries that B contains for term T.*

Proof: Consider the time when T was the current term. Recall that whenever the leader sends a new entry, it also sends the index and term of the previous entry.

Consider the first log entry that was added in the term. Either there was a $\langle$ term,index $\rangle$ match or not. If there was a match, then the entry sent by the leader is added to the log. This becomes the first entry in the term. If there is a mismatch, then the leader repairs the logs of the follower till both match till that point (beginning of term T). After that the first entry in term T is added. In both cases, for A and B the same entry is added as the first entry of term T.

Let us no prove by induction. Assume that the first k entries are the same (in term T). Consider the $(k+1)^{th}$ entry. It will be sent to both A and B by the leader. Both of them cannot add the entry unless their k^{th} entry in term T matches that of the leader. If there is a mismatch, then it means that there must be lost messages. In such a case the logs have to be reconciled. After reconciliation the $(k+1)^{th}$ entry will be added. It will still be the same entry.

Now, at the end of term T, if we find that A has more entries than B in term T (i.e., $N_A \geq N_B$), then we can say for sure that A has all the entries that B added. It must be that B missed some messages. □

Lemma 51 is a restricted version of the Log Matching. It only discusses matching in a given term. We now need to extend it to hold across terms. This is where we need to consider the way in which we conduct elections. Recall that the log of a candidate needs to be at least as up to date as the log of each voter. This means that its last entry must either have a higher term or a higher index. Let us consider the next lemma.

Lemma 52 *If two logs have the same entry at the same index, then the logs match till that index. The entry here comprises the command and the term.*

Proof: Consider the first term. This is true given the results of Lemma 51.

Now, consider the second term and the first entry that is added in it. Let us assume that the election was successful.

Now how did the leader of the second term get elected in the first place? It must have had at least as many entries in its log as a majority of the servers. This is a straightforward conclusion from the election restriction. Once it has been elected as a leader, it will bring all the logs into sync with it before they can add the first entry in the second term. If some server has additional entries that the leader does not have, then it will be forced

to delete them. Ultimately, for the first term, they will all have the same number of entries (also same entries) before they can add the first entry in the second term.

Now, let us extend this logic to the induction case. Assume that the Log Matching property holds for the first k terms. Consider the $(k+1)^{th}$ term. Here also a server cannot add its first entry unless the logs match with the leader for the first k terms. Only after that the first entry in the $(k+1)^{th}$ term can be added.

The only case that we have not considered up till now is the case when the election fails. This means that no server gets a clear majority. In this case, we pretty much waste a term. We move to the next term and again try to elect a leader until a leader is successfully elected. The wasted terms don't add messages to the log. The only modification that we need to make to the proof is that if the leader is elected in term k', then we find the latest term k that has a valid leader. The rest of the proof remains exactly the same.

We have thus proven by mathematical induction that the Raft algorithm preserves the Log Matching property. □

Given that the Log Matching property holds, we need to now prove that if an entry is committed by a leader it remains as a committed entry in the logs of all future leaders. This is, in principle, different from the Log Matching property because it brings in the notion of commitment, which we have hitherto not discussed.

Leader Completeness

This property establishes the finality of commitment. This means that once if an entry is committed, it remains present and committed in the logs of all future leaders. It is never forgotten in the system. Let us prove a lemma.

Lemma 53 *If a leader commits an entry, it is present in the log of the next leader.*

Proof: If an entry e is committed, then a majority of servers have stored it in their logs and also acknowledged the same. This means that for any leader to be elected, it needs to have a log for the current term that is at least as long as each of the voters. Given that the intersection of two sets that comprise a majority is non-null, there will be at least one voter that will have entry e. This entry needs to be present in the logs of the new leader as well for it to be elected a leader in the first place.

Hence, once an entry is committed in a term, it will be present with the leader that is elected next. □

Now, let us prove that no subsequent leader can delete an entry that has already been committed in a previous term.

Lemma 54 *If a leader commits an entry, then it is present in the logs of all subsequent leaders.*

Proof: We have already proven in Lemma 53 that if a leader commits an entry, then the next leader has it in its log. We now need to show that no leader is elected that does not have the entry.

Let us say that the leader of term k (L_k) commits the entry e. Without loss of generality, let us assume that term $k+1$ has a valid leader L_{k+1}. The reason that we can make this assumption is because if no leader is elected, then nothing happens in the term. Let us basically skip all the terms after term k in which no leader is elected. Ultimately, a leader will be elected. There is no harm in considering it to be term $k+1$ from the point of view of correctness in the proof. Now, as per Lemma 53, L_{k+1} will also have entry e in its log. What about the leader L_{k+2} of term $k+2$? Again with no loss of generality let us assume that L_{k+2} is a valid leader elected with a majority vote. The claim is that it will also have entry e in its log. The reason is as follows.

Given that L_{k+1} has e in its log, it will not remove it. Instead, it will ensure that the rest of the servers also add it to their logs if they don't have it already. We know that a majority of servers at this point already have e in their logs because it was committed by L_k and not subsequently removed. Of course, this process may not be fully successful because L_{k+1} may crash in the middle.

Now, consider L_{k+2}. If it already has e then there is no issue. Is it possible for it to not have e in its log? This is not possible because it will need a majority vote. There will at least be one server that has e in its log and thus it will not allow any candidate to get elected without having e. Now, L_{k+1} did not remove e from any log and thus the status quo was maintained. As a result, for the same reason that was used for L_{k+1}, L_{k+2} will also have e in its log.

We can thus prove by induction that all successive leaders will have the entry e in their logs. This means that once an entry is committed, it cannot be removed by any leader. □

Lemma 54 proves the Leader Completeness property. We thus observe that once an entry is committed, it cannot be removed and it continues to remain in the logs. We can prove an interesting corollary here.

Corollary 55 *If an entry is committed, then all the previous entries in the log are also implicitly committed.*

Proof: Lemma 54 proves that once an entry is committed in the remains in the logs of all leaders and also a majority of servers. Let us now use the Log Matching property. This means that in a majority of the servers including the leader all the entries preceding any committed entry are identical. Given that a committed entry cannot be removed (as we have just seen), and the process of removing entries starts from higher indexes and move to lower indexes, we can happily conclude that no entry that has a lower index that than of a committed entry will ever be removed. It is also present in a majority of servers. This basically means that it is *committed* and has achieved a degree of finality – it cannot be overwritten or removed. □

State Machine Safety

Given the previous lemmas that prove that Log Matching and Leader Completeness hold, proving state machine safety is quite trivial. It basically says that if a server has applied a committed entry present at a certain index in its log, then no other server will apply any other entry at the same index. This is true because if an entry at a certain index is committed, then all the servers will eventually store the same entry at that index. A majority of the servers already do, which is why the entry is committed in the first place. Even the rest of the servers that are somewhat slow cannot store any other entry at that index. This will violate Lemmas 52 and 54. Hence, a committed entry can be safely applied to a state machine. All other servers will also eventually do the same. Given that we are not considering Byzantine failures, the servers cannot do anything else.

6.5.7 Miscellaneous Issues

Cluster Membership Changes

In a practical system, servers can be added and deleted. This means that the Raft cluster membership has the potential to continuously change. A naive solution is to stall the system, change the system configuration and then restart the system. If a new server is added, then there is a need to copy the logs of the leader to the new servers that are added.

The advantage of Raft is that it does not suffer from any downtime. There is no need to pause or stall the system. It continues to work even when servers are added or removed from the system. The leader gets a

request to change the configuration of the system. Let the old configuration be C_{old}, and let the new configuration be C_{new}. This mechanism also allows us to join two Raft clusters.

Several problems can happen. While a larger cluster is being created, it is possible for two leaders to be elected. This is because the entire cluster has not joined yet to form one cohesive unit. All the servers may not even be aware how large the new cluster is, and they may not know what the majority mark is.

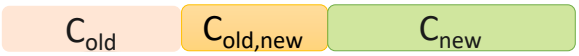


Figure 6.12: A joint consensus mechanism

A joint consensus mechanism is used by Raft to solve all such problems that arise when a new cluster is forming. Temporarily, a short-lived joint configuration is created called $C_{old,new}$. There is a need to replicate all entries to all the servers that belong to both the configurations. This is done as follows.

In the joint consensus mechanism, there is a need to elect or choose a leader. The leader of any configuration – C_{old} or C_{new} – may work as the overall leader. For any kind of majority-based agreement for elections and AppendEntry operations, there is a need to get votes from a majority of servers from both the old and new configurations, C_{old} and C_{new}, respectively.

The joint consensus mechanism is shown in Figure 6.12. Note that the joint consensus state is a valid state. At this point, client requests can be added to the log and can be serviced. Let us discuss the details.

Joint Consensus Mechanism

The leader of configuration C_{old} receives a request to convert it to C_{new}. It creates a new temporary configuration called $C_{old,new}$. This information is sent to all the servers as other regular messages such as heartbeat messages. At this point the logs are copied from existing servers to new servers. Next, the entry $C_{old,new}$ is added to the logs of each server. Once this entry is committed, this becomes the current configuration. During this period if there is a leader crash, the choice of a new leader is dependent on which configuration is used. It depends on the configuration that the candidate belongs to: C_{old} or $C_{old,new}$.

Once, the joint consensus has been setup and the entry $C_{old,new}$ has been committed, it is time to elect a leader for it. All the servers that have $C_{old,new}$ in their logs are eligible to vote. The temporary leader of the configuration $C_{old,new}$ then initiates the procedure to add and commits a new entry C_{new} that corresponds to the new configuration. Once this entry is committed and is sent to all the servers, the new configuration fully takes effect.

Log Compaction

Logs keep growing over time. It is important to ensure that the logs do not grow indefinitely. When a log reaches a predetermined size, we can take a *snapshot*. We can use the Log Matching property here. We consider an old index, record it and the store the rest of the logs that precede it in stable storage. We can discard the logs that were stored in the rest of the servers. No information is lost. The logs match anyway. Hence, storing one copy is just enough.

Sometimes the leader can transfer its snapshot to followers that are far behind. This fast-forwards them and brings their logs in sync with the leader.

Client Interaction

Raft provides linearizable consistency, which means that every operation appears to complete at a unique point between its start and end (more details in Section **??**).

Moreover, it is possible to quickly realize read operations. There is no need to add it to a log entry. However, there is a need to provide only committed values. The leader thus needs to commit an entry such that all the entries before it get implicitly committed. Then it is possible to quickly respond to a read request.

Chapter 7

Consistency

7.1 Consistency Models in Distributed Systems

Consistency models at a glance.

- Linearizability (atomic consistency)
- Sequential consistency
- Snapshot isolation
- Causal consistency
- Probabilistic consistency
- Eventual consistency
- Client-centric consistency
  - Read-your-writes
  - Monotonic reads
  - Monotonic writes
  - Session consistency
  - Bounded staleness

7.2 The CAP Theorem

Eric Brewer conjectured in 2000 that we cannot design a protocol that achieves some form of strong consistency, service availability and network

partition tolerance at the same time. At that point of time, this was an intriguing hypothesis. The community thought that we can indeed design a protocol that should be resilient to all kinds of failures. Of course, there are many definitions of these terms. A simple definition of consistency here is that a read fetches the value written by a write that has completed in the past (or a later write). The term *availability* means that every request is replied to in a finite amount of time. Partition tolerance refers to the fact that even if the network is partitioned into two, the system continues to work. Gilbert and Lynch [**?**] were able to formally prove this in 2002. The PACELC theorem first proposed in 2010 (and formally proven in 2018), built on this. It adds an element of the consistency-latency tradeoff, which CAP did not cover. A lot of this thinking was incorporated in state-of-the-art industrial designs.

For any real-world web service we would expect that all these properties hold. We would ideally not want to sacrifice any of these properties. However, life is not always that rosy. After all most databases provide the ACID properties, however, these are much more tightly coupled systems where the failure models are somewhat restricted. In a generalized distributed system it is hard to guarantee many such properties that one would normally expect in a data base or a file system.

Before proceeding, let us formally define these terms.

7.2.1 Basic Definitions

Atomicity

Before we start, a disclaimer is due. The term atomicity has multiple definitions and it varies from author to author. The particular definition that we will use is by Nancy Lynch [85], where she defines atomicity to be a property that is also known as linearizability [60]. Consider a distributed object like a simple shared variable (also known as a register), a database or a complex distributed data structure. All of them can be virtualized as a concurrent object that supports a set of methods that operate on them, change their internal state and return with a value.

Consider a simple distributed register (a variable). It supports two operations: *read* and *write*. The *read* operation returns with a value whereas the *write* operation changes the internal state of the register by overwriting its contents. It does not return any value. However, we can still make it return a status code indicating that the write was successfully completed. We thus observe that every function call has two components: a function

invocation and a function *response*.

A function execution is said to be *atomic* if it appears to execute instantaneously at a given point of time between its invocation (start) and response (end). This variant of atomic consistency is also known as linearizability. The system is said to provide *availability* if every request leads to a response unless the node generating the request fails. This basically means that the system cannot become unresponsive. Finally, we have *partition tolerance*. This means that even if the network is partitioned and a pair of nodes (across the partitions) become unreachable, the system as a whole remains responsive. It provides responses to requests and the responses are *correct*.

Theorem 56 *A basic read/write object in an asynchronous system cannot guarantee all of the following properties, when network partitions are allowed:*

- *Availability*
- *Atomicity*
- *Fair execution (all the requests are processed in a bounded amount of time)*

Proof: Assume that there is an algorithm A that provides all the three CAP guarantees. Consider a point of time where the network is partitioned. The set of nodes V in the network is partitioned into two non-overlapping subsets: V_1 and V_2 (resp.). Any message sent from node $u \in V_1$ to node $v \in V_2$ is dropped.

Consider a distributed read/write object x. If a write operation occurs in u ($\in V_1$) and later on the same object is read in v ($\in V_2$), then the write will surely be missed. This is because no message can be sent from any node in V_1 to any node in V_2. As a result, the information regarding the write could not have been transmitted. Let us look at this slightly more formally (on the lines of the original proof).

Let the value x_0 be the initial value of the read/write object. Consider an execution (sequence of operations) that only contains a to x by a node in V_1. This write sets the value of x to x_1 ($\neq x_0$). Let this execution be e_1. This write needs to complete because it cannot stall indefinitely (fair execution) and it needs to generate an acknowledgement or response (availability requirement).

Let us next consider an execution e_2 that only consists of a node in V_2 reading the value of x. We can assume that the execution terminates after this read operation.

Assume that there are no additional client requests after the respective operations in e_1 and e_2 finish. Let us now consider the combined execution $e = e_1 \mid e_2$ (first e_1 then e_2). Given that the network is partitioned, the nodes in V_2 will not be able to distinguish between e and e_2. This is because they will not receive any message owing to the write in e_1. If we only consider e_2, then the read should return x_0 (the initial value). However, if we consider the combined execution e and the atomicity property, then the read operation should return the value of the last write, which is x_1.

This situation is clearly contradictory in nature. We don't know what to return – x_0 or x_1. If we were to return x_0, then it will violate atomicity. If we return x_1, then it will violate causality because the value x_1 was never sent to nodes in V_2.

Hence, it is not possible to have any such algorithm A. □

This basically proves the CAP theorem. We clearly cannot guarantee/provide all three properties at the same time. It turns out that we can prove a very interesting corollary. Even if no messages are lost, but the network is asynchronous, it turns out that we cannot guarantee availability and atomic consistency.

Corollary 57 *The CAP properties cannot be guaranteed in an asynchronous network even if we assume that no messages are lost.*

Proof: Recall the FLP result where we had indicated that it is not possible to differentiate between a scenario in which a message is just "delayed" and it is lost. This had lead to a contradiction and we had proved that it is not possible to design an algorithm that provides the consensus guarantee. □

Part II

Services

Chapter 8

Distributed File Systems

File systems aim to provide a clean interface to users to access files on their local system. In addition, a file system also provides access-control mechanims to allow only authorized uses have access to the files and perform operations on files. For the former, the standard UNIX file system uses a data structure based on *inodes* and a hiearchy of direct and indirect addressing to store the contents of the files and related metadata. For the latter, the Unix file system uses the popular hiearchy of user, group, and others, and read/write/execute as permissions on files. For more details, we refer the reader to Bach [10].

Let us now consider recent planet-scale applications such as Facebook and YouTube. These applications allow multiple users to create, share, and access content simultaneously. It is estimated that each day, Facebook generates new data of the order of petabytes [1]. Similarly, YouTube adds content of the order of exabytes per day (https://www.wyzowl.com/youtube-stats/). Such applications, viz. Google Photos, TikTok, and Amazon are becoming more common in the present era. As can be noticed, the scale of the data that these systems have to store is tremendously large. Further, unlike the earlier decades of computing, today, most of the data and the users of the data are more likely to be geographically separated yet seamlessly connected via the Internet. For the organizations that own this data, this data is also a source of analytics leading to revenue generation.

However, multiple challenges have to be overcome to support both the user-driven functionality and the organization-driven requirements. Some of them include designing highly scalable systems for storing and retrieving the data, addressing the issues of latency and other network-induced phe-

[1] https://kinsta.com/blog/facebook-statistics/

nomenon, providing mechanisms and guarantees for concurrent data access by multiple users, and ensuring the availability and the durability of data.

The changing application requirements necesstitate a fundamental rethink on file systems. Traditional file systems optimize for parameters such as the block size, extendiblity of the file size and supporting such seamless extension (see also Question 8.1), read and in-place write operation, and access control mechanisms. This conventional wisdom is now up for a revision due to multiple factors including the decreasing cost of disk drive per GB of storage, reducing latency for each operation, minimizing the number of disk accesses needed for any opeation, less or no reliance on in-place write, versioning control, and the like.

The rethink on the conventional wisdom mentioned in the earlier paragraph require the design and implementation of distributed file systems that go beyond the notion of traditional file systems (such as the Unix file system, ext3, Windows NTFS) and are optimized based on the new and emerging application requirements.

In this chapter, we first review some of the common design features of distributed file systems. Subsequently, we provide details of some of the popular distributed file systems starting from NFS [104] to Colossus [61]. We also explain in brief the file systems that other platforms, Haystack [17] and Tectonic [94], that Facebook uses. The detail we provide covers some of the design choices and assumptions, operations, and some implementation details. For simplicity, we do not distinguish heavily between file systems and object stores. While both sastisfy a similar set of requirements, the latter are seen to be more application-specific. Being application-specific allows such systems to introduce optimizations driven by the target application.

8.1 Common Design Features

The origin of distributed file systems can be traced to projects such as the NFS [104] from Sun Microsystems and the Andrews File System (AFS) [62] of the Carnegie Mellon University. The basic idea of these distributed file systems is to allow the users to use remote computers also to access and store files as if they are stored locally. Some of the common design features of distributed file systems are as follows.

- **Durability:** Data durability is a key concern for any distributed file system. This refers to the idea that the data can be recovered in case of failures to the underlying hardware. As distributed file systems use multiple hardware resources so as to provide scalability, hardware

errors due to device malfunction may corrupt data. Distributed file systems usually employ a form of replication to safeguard against such issues. Replication however comes with a set of challenges invovling the creation/ updation/ maintenance/ and consistency of the replicas. Distributed file systems have to therefore understand how to provide support for these mechanisms and their implication on the file system usage.

- **Availability:** Distributed file systems have to contend with possible errors from the users or the file system itself. These errors are not the result of poor system design and implementation but a byproduct of the intended usecases and scale. Distirbuted file systems cater to clients that are spread over the network and naturally, the requests are received and served over the network. Delays and other network induced vagaries affect the file system and its operations. Similarly, as the file system scales to beyond petascale and zeta scale storage, the number of components that paly a role in the entire system increases. This increase will invariably result in an increase in the number of failures. Hence, distributed file systems have to provide for error handling mechanisms and the necessary semantics to recover from errors.

- **Consistency Model:** Distributed file systems should naturally support concurrent operations on files and simultaneous sharing of files by users. This requires the distributed file system to provide mechanisms that allow such concurrent operations while providing reasonable guarantees on the outcome of the concurrent operations. This suggests that distributed file systems have to specify and operationalize the semantics of the concurent operations.

- **Scalability:** One of the advantages of using a distribtued file system is to provide for scale as the storage requirements grow. To this end, distributed file systems should be designed in a way that allows for scaling up the capacity in a seamless and transparent manner without degrading performance.

- **Namespace:** The amount of metadata and the data structures that a file system maintains is dictated by how the system handles the file and directory namespace. The mechanism that the filesystem uses to search for files and update the metadata is based on the data structures that the file system sets up. Further, memory system considerations such as where the metadata is stored, the cost of accessing/updating

the metadata, costs associated with keeping the metadata on stable storage, and the like influence the organization of the namespace.

Some distributed file systems use a flat namespace to simplify handling of metadata. This decision is usually a result of aiming for lower latency and higher throughput.

- **Block Size:** Blocks are the unit of storage of files in a file system. The size of a block has a bearing on the size of the metadata that the file system has to keep. In addition, block size also influences the network traffic in a distributed setting, internal fragmentation, cache size and behavior, among other things. Unix file systems typically use a block size of 0.5 KB to 4 KB.

  Depending on the application that distributed file systems target, some use a very large block size of several MB. This allows such file systems to scale to large files while not increasing the amount of metadata. Observe that a small block size prevents internal fragmentation at the expense of more metadata, while a large block size can result in internal fragmentation but reduces metadata.

Beyond the ones mentioned above, there are application-driven concerns such as optimizing for bandwidth, latency, number of files, typical size of files, and the like. The functionality, semantics, system assumptions and support vary across distributed file systems as we will see below.

In this chapter, we first set out to understand the challenges and design space of distributed file systems with respect to the above mentioned properties. This exercise will help us understand the choices that system-designers have in arriving at designs of distributed file systems. Subsequently, we will see the choices made by some of the proposed distributed file systems over the decades.

8.2 The Design Space of Distributed File Systems

One of the fundamental properties of distributed file systems over a traditional file system is to provide support for multiple users to access a set of files over a network, subject to access control. Figure 8.1 shows the typical system abstraction at a very high level.

Consider a typical use case of the system depicted in Figure 8.1. A client wants to access a file and has to contact the server for this purpose. The client has to supply information that allows the server to locate the file that

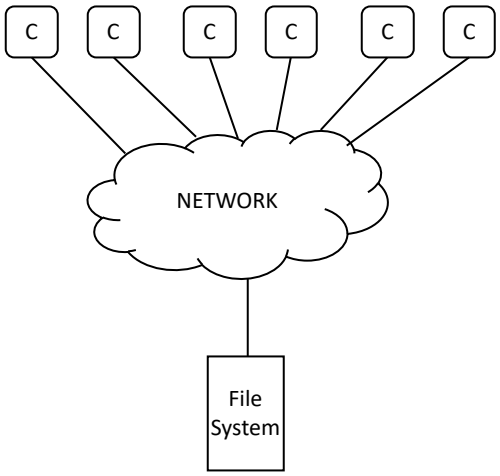


Figure 8.1: Figure illustrating the simplified abstract view of a distributed file system. The entities written as C indicates a client.

the client needs. This is usually specified as a pathname if the files are organized in a hierarchical manner. Modern object based storage systems choose to use a flat namespace and each object is given a unique identifier. In this case, the server locates the object with the help of additional data structures.

Once the access model is set up, we now understand the choices in how the client performs other operations such as read/write on the file. There are multiple choices and we detail some of the choices. One choice is for the server to get information from the client on the part of the file that the client wants to access. The other choice is for the server to transfer the file contents, patially/in-full, to the client and let the client work on a local copy of the remote file. In the former case, the system resembles a client-server system with the server being responsible for managing the file system and the contents of the files. This solution is possible when one considers situations where the network bandwidth and latency considerations allow for the server to play a major role in every client operation. In the latter case, the server allows the clients to cache the file in part or in full. The client can then perform operations on the file as if it is a local file. At the end of these operations, the client has to inform the server if the operations change the contents of the file or the metadata related to the file. This model works best in settings where the network bandwidth and throughput are low. After the inital transfer of the file contents, client operations on

the file behave similar to operations on a local file.

Maintaining caches of files being used at the client side however introduces a level of indirection[2]. Even as caches improve locality, and in this setting reduce network traffic, they introduce yet another problem that is particular to any distributed system in general.

In a simple setting, subsequent to making any updates to the local copy of the file, when the file is to be closed, the client can contact the server to send the updated file contents. The server will make the changes to its (original) copy and send an acknowledgement of the same. Often, network bandwith limitations may hamper the throughput of the file system and can introduce delays in when the server can acknowledge the completion of a file udpate at the server end. The client has to wait for this acknowledgement before moving to the next instruction. To reduce this delay, the file system can instead send an immediate acknowledgement of the file update request from the client, pool multiple write operations, and update the file contents once for the entire pool. This model is called as the *asynchronous update* operation. While this improves on the latency of the update, it poses dangers with respect to loss of update if the server crashes without applying the update. To address this situation, one solution is for the client to treat the acknowledgement from the server as a weak guarantee, keep the data in its local buffers, poll the server for the status of the update. The client can release the local copy only after it knows that the update is applied successfully at the server.

Even with caching, the above is indeed a very simple view. Consider that a distributed file system should naturally allow for multiple clients to share files. In this situation, multiple clients will likely access the same file and perform operations on the files concurrently. In this direction, imagine that two clients access the same file `foo.txt`. When the server allows for parts or the entire file to be cached at a client side, each of the clients may be caching a common portion of the file. Each client is unaware that there may be other clients having the same (portion of the) file also in their cache. Now, if both the clients write to common location of the file, each client will be unaware of the changes made by the other client. Figure 8.2 shows depicts the situation.

The situation demonstrated in Figure 8.2 is symptomatic of concurrent writes to any shared object. As an example, suppose that the file that two clients access concurrently corresponds to a source file in a large project.

[2]See the quote attributed to David Wheeler, "All problems in Computer Science can be solved by another level of indirection"

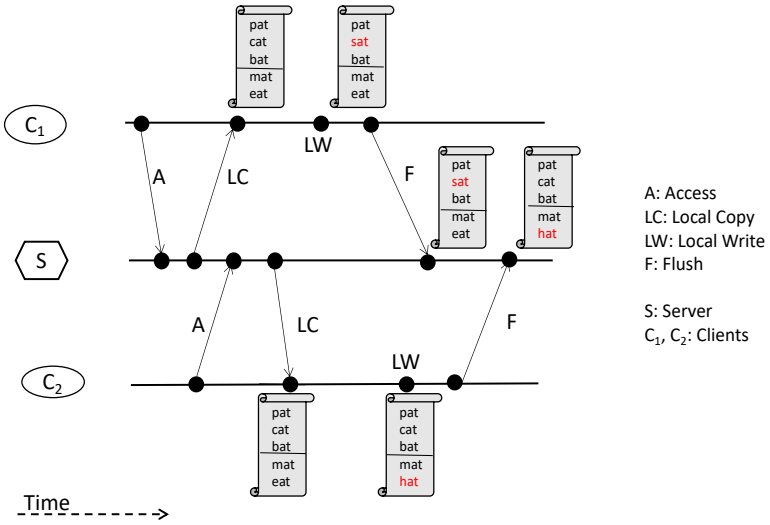


Figure 8.2: Figure illustrating a possible clash between writes by multiple clients.

If these writes overlap, it is not clear as to which one would eventually correspond to the final copy and how to reconcile the differences. Think of systems like `git` and the way they deal with such concurrent updates. In order to distributed file systems have to formalize the semantics of what happens in such concurrent writes. Notice that concurrent reads pose less of a problem compared to concurrent writes.

Surprisingly, there are several ways to formalize the semantics of concurrent updates to a file in a distributed system. Depending on the application need, there exists a spectrum of design choices in this case. The semantics are often called the consistency model of the distributed file system. At one end of the spectrum, a distributed file system may simply forbid sharing of files acros multiple clients at the same time. Since there is no concurrent access to files, the server is effectively able to serialize all the updates to the files – in the order the files are accessed. This is very limiting and goes against the grain of a distributed file system. Another possible solution is for the server to allow concurrent access to files but make no guarantees on the outcome of the concurrent write. Referring to the earlier example, the source file may have an inconsistent state after the two writes are affected. This solution is not appealing but has no overhead at the server.

Fortunately, better solutions exist. The server can implement a policy often termed as the “Last Write Wins”. In this case, the contents of the

file (in full or partially) that were the last to be acknowledged as successful by the server will be the contents that will be served to later clients. REWORD. This consistency model is similar to a serialized model of updates. (Question: Identify the serialization point for every write). There are other consistency models also such as weak consistency, eventual consistency, strong consistency, that are also possible from a system standpoint. In addition, depending on the nature of applications that the filesystem targets to support, the system may provide better consistency semantics for specific operations.

So far, we have worked under the assumption that the file updates coming from clients will also come with an offset specified. The POSIX style write operation has the file offset as one of the input parameters. On the other hand, in particular with emerging application scenarios, what is more important is to have light-weight guarantees on adding the data to the file whereas the offset where the data is written may not be as important. Examples include log files where usually the timestamp of the log entry help assign a timeline to the data and from a user point of view, having the log entry in the file is more important. In line with this observation, a distribtued file system can choose to target specific operations and provide stronger consistency guarantees on those operations. The operations can be chosen by the file system based on the applications that the file system targets.

Faults:

The earlier paragraphs did not cover any discussion on faults. Distirbuted sytems in general have to also consider aspects of fault tolerance due to the their nature. Failures of the network, storage hardware, or client/server machines have the potential to introduce multiple challenges. For instance, if the distributed file system has a single server, crashes of the server or instances when the server is unavailable due to network issues, introduces difficulties with respect to the *availability* of the overall system. In a single-server system, a server crash also renders stateful server based models open to loss of state. This requires systems to provision for mechanisms for server state recovery and addressing the system availability in the interim.

Such mechanisms include a combination of techniques such as provisioning for a shadow server, supporting only for a subset of operations in the interim, commiting server state and updates to the server state to durable storage before acknowledgements, checkpointing, having multiple servers, and so on. It must be noted that some of these mechanisms have been

found useful in other distributed system designs also.

Faults of the storage hardware in the context of distributed file systems have the potential to create loss of user data. A standard mechanism to handle this is to create replicas of the data. In a solution that uses replication, the system divides each file into logical pieces and replicates each piece by a chosen factor. This protects against data loss and corruption provided all replicas are maintained as identical copies. Doing so however requires additional overheads when the file contents are being updated. For every file update, for the corresponding piece, each replica has to be updated. Notice that each replica may be on a different physical device and these update operations may also fail while some of these succeed. This creates further complications that systems have to solve. Systems have a choice between targeting higher throughput and allow for inconsistencies in the file contents versus aiming for stronger consistency guarantees and possibly affecting througput.

Another fault that arises in the distributed file system is about network faults that may potentially induce partitions in the set of clients and the server(s). In this situation, a server may be temporarily unable to reach some set of clients or vice-versa. Such a situation may result in stale information at server or the client while messages from one do not reach the other.

In the above, we discussed multiple use cases and challenges around a distributed file system. Other design considerations surround the data structure(s) that the file system maintains to store metadata about the files. Recall that the Unix file system has an elaborate three level data structure consisting of a file desriptor table, the file table, the per file inode table, that promote a great deal of flexibility in how the Unix operating system handles files and directories. These data structures allow the Unix file system to maintain a hierarchical namespace.

Even as the hierarchical namespace works well for generic file systems, the benefit of the hierarchical organization is often offset by other application-specific requirements. For instance, when the system has to only deal with specific kinds of files, such as images or videos, certain optimizations are possible. For instance, a set of such objects can be stored in a single file in the file system with offsets indicating the position of each object in the combined file. Similarly, it may not be worthwhile to support write operations on image or video data. However, changes to image or video can be instead supported by a versioning mechanism that stores the full content for each change. Doing so also reduces the time needed for a server to serve any version of the file instead of preparing the required version through the change log. Notice that the two possibilities depicted here present a trade-

off in terms of the amount of storage needed versus the latency of the access operation at the server.

A related aspect that goes along with namespace considerations is about the design choice of what metadata the system keeps about files. The traditional Unix file system keeps information such as access control and access timestamps as part of the metadata. As application specific distributed file systems cater to specific file classes, they do not need efficient support for particular operations. In such cases, it is possible to forgo some parts of the metadata to gain efficiency even while losing out on ceratin other operations. The specific pieces of metadata that the system can skip may be guided by the needs of the application.

The concerns in those decades corresponded to making fficient use of the available storage space and limited throughput of the disk drives. Of late, the cost of storage drives has plummetted to AAA per GB of storage.

8.2.1 Sample Realizations

In the above section, we have described multiple possible choices for each of the design aspects of distributed file systems. Now, we will provide quick examples of how some of the extant distributed file systems fare in this design space and the choices they make.

A simple example is the SUN NFS [104] system which acts mostly as a facilitator for clients to use a remote file system. It uses a single server and does not provide any replication on its own. It also is stateless and hence provides no gurantees on how the updates are visible to other clients. It however allows clients to keep blocks of the file in their local cache and push updates to the local cache before sending the udpates to the server. The Andrew File System (AFS) [62] is very similar to NFS except allowing for the clients to keep the entire file as a local copy.

The post-Internet era distributed file systems such as the Google File Systems [51], Colossus [61], and Tectonic [94] are examples of highly scalable file systems that use replication, provides for avalability, guarantee a notion of consistency, and so on. An example of an application specific distributed file system is the Facebook Haystack [17] object store that trades-offs metadata for faster access to the files.

More details of the above systems are presented in the sequel of this chapter.

8.3 Network File System (NFS)

NFS [104] started as a project at the Sun Microsystems in 1984. As with any file system, NFS too supports typical operations on files such as open/ read/ write/ close, and seek, and on directories such as create file/ make directory/ rename file/ rename directory/ delete file, and delete directory. The idea is to create a consistent namespace for files irrespective of the location of the file, whether the file is on a local machine or on a remote machine.

The first version of NFS, NFS v1.0, is an in-house project. The second version, NFS v2.0, is what is released for public use.

8.3.1 Goals and Assumptions

NFS aims to provide a machine and system independent interface and transparent access to the remote file system. With the popularity of Unix at that time, NFS adopts a Unix style interface to its calls and semantics. In addition, NFS underscores the fact that distributed systems are prone to failures, and hence the file system should be able to recover from server and client crash failures, and network failures.

NFS assumes that most file operations will be to read and there are fewer write operations. This assumption is validated by the statistics of that era collected by Sun Microsystems [104, Appendix 2]. In addition, NFS assumes that there is little sharing of files across users and the predominant usecase is concerned with users accesing files stored at a remote server. NFS therefore aims to optimize its performance on single user operations. This design model corresponds to user-oriented design. Early versions of NFS did not support locking of remote files too – an operation that would be required more in a setting where multiple users share a file.

8.3.2 Operation and Semantics

NFS uses Remote Procedure Calls (RPC) to support remote file operations. There are two main components in the implemenation: a server and a client. A server program runs on a machine that stores and serves files, and a client program runs on the user's machine. NFS relies on network calls and transfers each client request to the server. Figure 8.3 shows the basic architectural diagram of NFS.

The protocol is stateless, synchronous, and blocking. The server program does not store any details of past accesses of clients. So, every client request

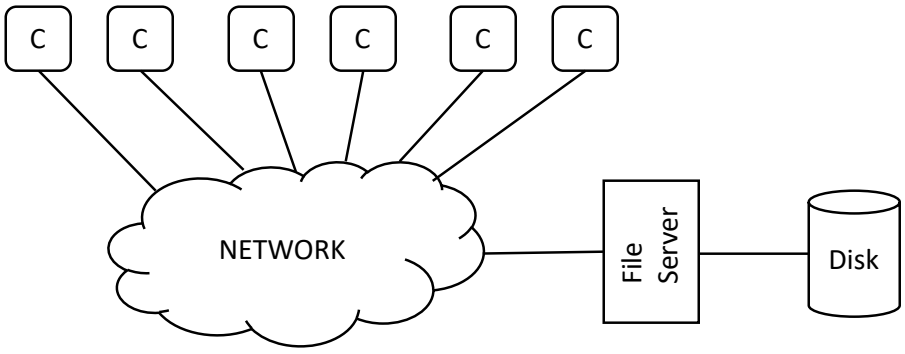


Figure 8.3: Figure illustrating the overall design of NFS. The entities written as C indicates a client.

has to include all the parameter values needed to complete the request. The client program waits until the response is obtained from the server. Having a stateless server model simplifies some of the fault recovery issues such as (i) a server crash handled by the client by a retry, and (ii) a client crash need not be handled by the server.

NFS used the TCP/IP suite of protocols for its network communication and uses the UDP protocol with its concomitnant possibility of packet loss. However, this is not a limitation since the protocol is stateless. Any packet losses will be handled by retry.

To maintain the statelessness of the server, the server has to ensure that all modifications to the file state are committed to a stable storage before returning results to the client. In addition, as the server is stateless, there is no provision to support file locking of remote files. Such lock information would require the server to keep track of some state information. A fallout of the lack of support for locking is in the lack of consitency guarantees offered by NFS on concurrent write operations. Two clients that issue a file write request to the same file can see inconsistent results.

In the simple design that is apparent from the above discussion, it appears that every client request has to necessarily reach the server for completion. This naturally limits the performance that NFS can achieve. To address this problem, NFS uses caching at the client side and the server side. This caching allows the client to cache blocks of files and make modifications at the client side instead of sending these as a request to the server. The client can then push all the updates when the client is done using the file. This is termed as `flush-on-close` semantics. Many file operations tend to also query attribute information on flies, such as the `stat` command

in Unix. These operations do not modify the contents of the file. Hence, NFS keep a cache of the file attribute data on the client for open files. This step also reduces the need to contact the NFS server for every file operation. Time limits are used to refresh the attribute data so as to keep such data current. NFS server also maintains caches of the inodes, buffers, and directory lookup, so as to gain performance on these operations.

The above indicates the trade-offs between performance and consistency issues introduced by caching. Two clients that read the same file at the same time may be getting different results based on what these clients have in their caches.

Using stateless protocols also necessitates another common practice that is used by most stateless protocols in distributed computing. All operations are idempotent in nature. This is ensured by having, for instance, unique identifiers on files (and directories). If a client wants to delete a file named "test" with an identifier of "0x18370399", then the delete operation is issued as `delete(0x18370399)` and not as `delete(test)`. (see Question 8.3 at the end of the chapter on why the latter is not idempotent and potentially risky.)

Design Features of NFS 2.0

NFS essentially acts as an interface between an existing file system and a client accessing the file system via a network. Therefore, NFS also does not have any in-built mechanisms to address the scalability issue. For the same reason, NFS does not have to make any choices with respect to features such as the namespace and the block size. These will be the corresponding mechanisms that the underlying file system uses.

Ensuring durability also is left to the underlying file system. If, for instance, the underlying file system uses techniques such as RAID, then the benefits of such techniques carry over.

With respect to availability, notice that NFS adopts the stateless model so that failures are handled by retries. So, NFS does not by itself have any mechanisms to deal with failures and improving availability.

NFS also assumes that most files are not used in a shared mode. Therefore, it does not provide any guarantees on the outcome of concurrent read and write operations. As NFS allows clients to cache data, it is possible that two clients that both cache a file may be getting different results on a read operation to the same offset depending on what is cached at each client.

8.3.3 NFS v3 and NFS v4

In the above, we largely discussed NFS v2.0. Later versions of NFS, in particular, NFS v3.0 (cf. IETF RFC 1813) and NFS v4.0 (cf. IETF RFC 7530), added more features on top of NFS v2.0. We breifly mention those here.

The NFS v3.0 server is still stateless, but is designed to use fewer network operations, achieves a faster write, and aims to reduce server load. A particularly new operation in NFS v3.0 is the *asynchronous write* operation. In the semantics of asynchronous write operation, the client sends the write data to the server and the server acknowledges the receipt of the data. However, the server need not write the data to stable storage before sending the reply. At the server end, asynchronous writes provide the server with several options so as to determine the best policy to synchronize the data. The server may never schedule the write or may wait for multiple writes to be performed together. The latter option inherently supports better buffering and parallelism. The client issuing an asynchronous write operation keeps the copy of the data in its cache. When the client wishes to release the data from its cache, it polls the server with a COMMIT message to check if the write is completed. In this case, the server sends a positive reply only if the data under question is written to stable storage and sends an error otherwise. The client, on receiving an error message, can resend the data asynchronously.

NFS v3.0 also added a rudimentary locking feature of remote files via an ancillary protocol called the Network Lock Manager (NLM). Calls to lock/unlock files are prepared as RPC calls to NLM. Unlike NFS v3, notice that NLM has to be a stateful protocol since the information on which process has locks to a particular file is to be maintained. The NLM protocol also has particular mechansims to address a network partitioning, a client crash and a server crash.

NFS v4.0 is a stateful protocol unlike the previous versions of NFS. In addition, NFS v4.0 introduces three key functionalities: Compound operations, leases, and locking. Compound operations allow a for combining a sequence of operations to a single remote operation thereby reducing network traffic. In this case, atomicity of the entire set of operations is not guaranteed. On the first error that the server encounters in a compound operation, the server does not execute the remaining operations in the comound operation and replies with the results of the successful operations.

Finally, NFS v4 allows for clients to lock remote files and this support is part of the protocol instead of delegating this support to auxiliary protocols

such as Network Lock Manager (NLM) that NFS v3.0 does. The server in NFS v4.0 being stateful can keep track of the lock information on files. Locking is coupled with leasing where the server grants a lease of a certain duration to the client. The duration of the lease is set at installation time and is usually of the order of few tens of seconds (60s to 90s is a fairly commonly used value). The leases are automatically renewed as the client performs any active operation on the files. If the server experiences a crash and comes back up, the NFS v4 server requires clients to reestablish locks within a given time window. During this time window, service is interrupted.

Design Features of NFS v3.0 and NFS v4.0

NFS v3.0 and v4.0 still essentiall acts as an interface between an existing file system and a client accessing the file system remotely via a network. Therefore, these versions of NFS also does not have any in-built mechanisms to address the scalability issue. For the same reason, these versions of NFS does not have to make any choices with respect to features such as the namespace and the block size. These will be the corresponding mechanisms that the underlying file system uses.

In NFS v3.0 and v4.0, ensuring durability also is left to the underlying file system. If, for instance, the underlying file system uses techniques such as RAID, then the benefits of such techniques carry over.

With respect to availability, notice that NFS v4.0 adopts a stateful model. Hence, it includes some steps to address the availability issue. The NFS server maintains the state of files and the owners of locks on these files. This state information needs to be recreated if the server crashes and recovers. Service interruptions and lack of availability can ensue in this interregnum.

NFS v4.0 in particular uses a stateful protocol and supports locking. It appears that this allows for a more stricter consistency model for concurrent writes. However, notice that the exact consistency achieved will also depend on the NFS server implementation and the support that the underlying file system provides. For instance, advisory locks on a file by a client still allow for writes by other clients. RFC 5661[3] provides more information on advisory locks and mandatory locks in the context of NFS v4.0.

[3]https://datatracker.ietf.org/doc/html/rfc5661

8.3.4 Summary of NFS

From the above, we conclude that NFS introduced a pioneering idea to the design and development of distributed file systems. NFS works well under the assumption that most files are not shared, very few concurrent operations exist, and most operations are read based, and the like. In such a setting, a simple consistency model coupled with client and server side caching helped NFS address issues such as performance, scalability, and consistency. However, the NFS server being invovled in all operations on every file does become a bottleneck. Subsequent designs of distributed file systems remove some of these limitations and also change the set of assumptions in part owing to changes in the landscape of dstributed computing.

8.4 The Andrew File System (AFS)

In this section, we describe the Andrew File System (AFS) and its development and features. The Andrew File System started as the ITC file system in 1985 with a subsequent new version termed as the Andrew File System, AFS v2.0 or just AFS. The ITC distributed file system is often termed as AFS v1.0. We will describe both the versions, starting with v1.0.

AFS, just like NFS, assumes that most files are not shared across users. It also assumes that most files are read in full and sequentially by applications/clients. These assumptions drive the design of AFS to optimize for these operations.

8.4.1 AFS v1.0

AFS v1.0 allows clients to access files on a remote file system just as they are on a local disk. To create this transparency, AFS supports a mechanism called whole-file caching unlike NFS. In this mechanism, the first time a remote file is opened by a client, the entire file is transferred to the client side and kept locally by the client. This allows all subsequent file operations to be performed locally. On the file close operation, the entire file is again sent over the network to the server. This simple description misses a few details which we elaborate now.

AFS uses the namespace structure of the underlying (remote) file system. Clients have to specify the full pathname to access a file. The AFS server reads through the file system hierarchy and transfers the entire file to the client so that the client can keep a local copy. Further read() and write(offset, data) operations do not go through the server. Each operation however

follows after the client checks the validity of the file from the server. On a client close() operation, the client checks if the contents of the file changed between its open and close operation. If so, the client sends the entire file to the server for updating the remote copy.

If the client still has the file in local space, a subsequent open() operation need not fetch the entire file always. The client can check with the server if the file changed in the interim: if not, the local copy can be reused.

The simple design of AFS v1.0 however has a few drawbacks. Primary among them are the contention at the server, cost of operations on the server including the cost of replying to the client call to check the validity of the file(s) they own locally, load imbalance issues, and the limitations of handling a large number of clients. In the early version of AFS v1.0, the server could engage with up to 20 clients. [**?**].

8.4.2 AFS v2.0

One of the ways to reduce the number of calls from the clients to check the validity of the file that they hold locally is to turn around the initiator of the check. In other words, if files are not usually shared often, it is safe to imagine that there would be fewer instances when the client call to validate the status of a file returns a response from the server that corresponds to indicating that the local copy is valid. Thus, several calls of this kind are redundant and offer a scope for improvement.

In AFS v2.0, the server sends an invalidate message to clients indicating that a copy of the file that they own locally is no longer valid. This mechanism, essentially like a push instead of a pull, requires the server to send a message to clients that hold a copy of the file whose contents change. This mechanism requires the server to maintain the required state information in terms of the list of clients that have a copy of a given file. Each client will presume that the local files that it holds are valid unless it hears from the server otherwise. AFS v2.0 names this mechanism as a **callback**.

Another optimization that AFS v2.0 introduced beyond v1.0 is to provide each file with an identifier in addition to the pathname.

8.4.3 Consistency

AFS follows the "last write wins" consistency model where the updates of the last client that performs a write is what is visible to other clients. The above is true so long as the clients are on different machines. However, if the clients are on the same machine, the updates of one client are immediately

visible to the other client. The latter model essentially resembles what typical UNIX based file systems follow.

There is one difference between how NFS and AFS treat file updates. In AFS, the entire file has to be sent back to the server at the time of close whereas in NFS, only blocks of a file are sent back to the server at the time of close. This distinction also happens to match with how NFS and AFS differ in how they cache files locally. Notice that this distinction also has some implications for how the files get impacted due to writes. If multiple clients modify parts (blocks) of a file in non-overlapping areas, it is not always clear that the updates can be merged. For instance, if the file corresponds to an image, and if all the updates are taken together into a common file, clients (users) will see portions of the file that may not be a valid image file or an expected image file. Hence, in some settings, it is important to write back the entire file as in AFS as opposed to only blocks in NFS.

8.4.4 Fault Tolerance

Like NFS, AFS has mechanisms to recover from server and client failures. If a client is unavailable while the server sends a invalidate callback, the client would not be aware of such messages. Therefore, upon recovering from any errors, clients have to treat all their local file copies as invalid and seek the validity of files from the server.

Similarly, clients have to handle a server crash by again checking for the validity of the files that they have a local copy of. The server maintains state information corresponding to the list of files that each client has open. However, this information is kept in the volatile part of the server memory and is susceptible to information loss on a server crash. Thus, the server may not be able to recreate the required state information on a crash. One possibility is for the server to immediately notify on recovery that all client held copies are invalid.

8.4.5 Design Features and AFS

We notice from the above discussion that AFS is not very different in its features with respect to NFS. The AFS server acts as a layer between the clients and the underlying remote file system. Therefore, features such as durability, metadata, and blocksize follow from the corresponding features of the underlying file system. The consistency offered by AFS is essentially the last write wins for clients on different machines. Compared to NFS, as the entire file is copied back to the server during a close operation where

the file has changed, the last write wins is applicable to the entire file and not a block of the file. The AFS protocol handles faults via suitable retry and invalidate/validate mechanisms. AFS, both v1.0 and v2.0, suffer from scalability issues due to the large amount of messages between the clients and the server. While AFS v2.0 improves on the scalabilty compared to AFS v1.0, both fail to achieve the desires of scalability for current generation systems.

8.5 The Google File System (GFS)

From the section on NFS, we understand that it is possible to make some gains in performance by trading off on consistency. However, a few deep issues exist in the design of NFS. Every file write operation eventually requires action from the server. This will become a bottleneck as the system scales. Further, each file read operation invovles multiple disk operations that may impact the performance of the file system. To be able to process the requests of multiple clients, one also needs a powerful server supported with other software mechanisms such as the ability to handle multiple connections, load balance, fault tolerance, and so on. The presence of the server also makes it a single point of failure.

In addition, the nature of workloads and system assumptions changed over the early 2000s. The rapid inroads made by internet and online platforms for shopping, banking, search, and entertainment, generated large volumes of digital data. It is estimated that organizations such as Google and Facebook produce and process data of the order of a few petabytes every day.

This large volume of data and its processing created workloads with very different characteristics compared to the earlier decades. For instance, appending to a file has become a useful operation instead of the standard write operation. In the former, the location of the write is not specified by the user so that the file system can identify a suitable location to add the data specified by the user, whereas in the latter the user specified the offset (location) and the data. Another such change in the nature of workloads is the use of concurrency and parallelism. The large volume of data that workloads deal with requires recourse to parallelism. Workloads may also produce huge amounts of data as a by-product. The file system should therefore be able to support concurrent operations on files with appropriate consistency gurantees.

There have been tremendous changes to hardware too in the post-2000

era. It has become easy and cheaper to prepare systems with off-the-shelf commodity machines rather than invest in expensive high-end hardware which also becomes obsolete over time. However, off-the-shelf commodity systems come with a higher rate of component failure, and hence system software has to be prepared to deal with component failures.

The above considerations led to the design of the Google File System (GFS) in 2005 [51]. The design principle can be termed as *workload-oriented* design where the main consideration is to cater to the nature of most frequent workloads that use the distributed file system. The Google File System therefore incorporates an operation called `record-append` whose semantics is different from the `append` operation supported by Unix style file systems. In the latter, the `append` operation adds the data to the end of the file poistion – the position that the client believes to be the end of the file. In GFS, the location where the data is written is chosen by the system and the system provides certain guarantees on the append operation (see also Section 8.5.1). For this reason, GFS is *not* POSIX compliant.

8.5.1 Design and Implementation

The GFS design includes three main components: a master server, a set of chunk servers, and the client. One of the main design principles of GFS is to not require the server (the master server) to be playing an end-to-end role in any operation. To this end, the server keeps only metadata related to the files and does periodic system monitoring. All client operations are offloaded to the chunkservers after a quick handshake at the server.

Basic Design

GFS stores files using fixed-size *chunks*. Each chunk has a unique 64 bit identifier called the *chunk handle*. Chunk servers store the chunks on their local disk and each chunk is replicated on multiple chunk servers for purposes of reliability. Note that a commodity machine can act as a chunk server and hence prone to component failures. Replication of chunks across multiple chunk servers addresses this aspect. Figure 8.4 shows a schematic architectural view of GFS.

The metadata that the master server maintains includes information related to the file namespace, access control information, information about the chunk handles of the chunks pertaining to the file, and the location information of the chunkservers that store these chunks.

For every chunk, one of the replicas storing the chunk is also marked as

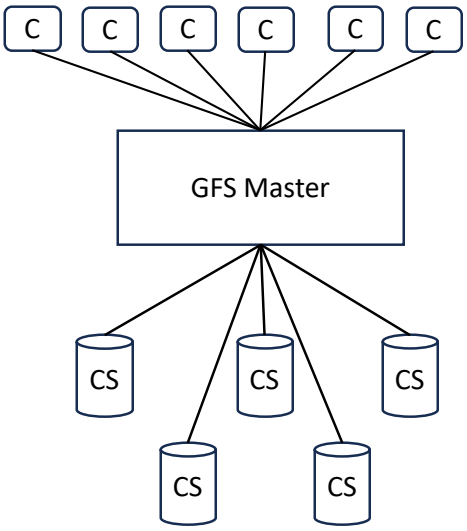


Figure 8.4: Figure illustrating the overall design of GFS. The entity marked with 'C' corresponds to a client. The entity marked with 'CS' corresponds to a chunk server.

the *primary* chunk server and others as the secondary chunk servers. The GFS master server nominates the primary and the role is assigned as a lease. The master server monitors the health of the primary chunk server so that in case of failures of the primary chunk server, the master server can reassign the role to another chunk server containing the chunk. The master can also reassign the role of the primary for a chunk to another chunk server at the end of a specified lease period.

Operations

We will now illustrate how GFS performs operations such as read/ write/ and record-append.

- **Read:** A read operation starts with the client issuing a read request indicating the file name and the offset to be read from. On receiving this information, the server replies to the client with the chunk handle location(s) of the chunkserver(s) holding the chunk corresponding to the desired offset. With this reply, the server essentially offloads the read operation to the appropriate chunk server and does not have to play a role in the actual transfer of data.

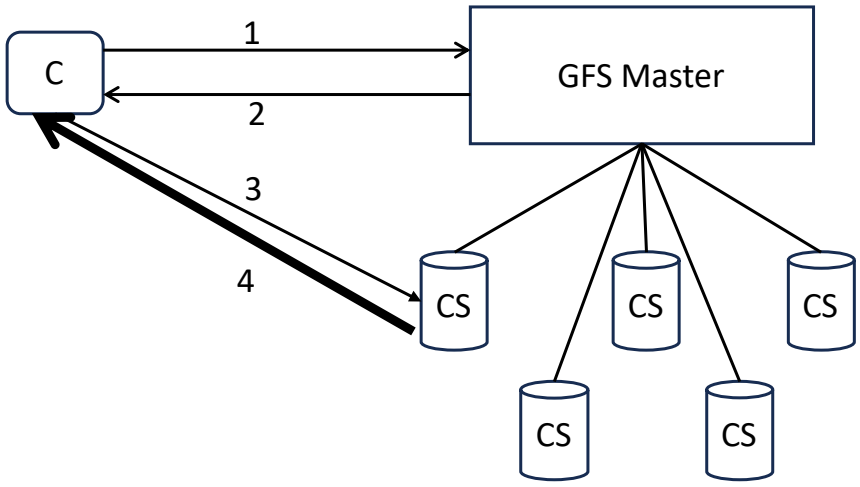


Figure 8.5: Figure illustrating the steps in a read operation in GFS. CS refers to a Chunk Server. Thin lines indicate transfer of control/metadata information and thick lines indicate transfer of data.

The client contacts the chunk server and provides information on the starting and ending byte numbers to be read. The chunk server can reply to the client with the corresponding data.

Notice that the GFS master server is only replying with metadata information whereas the actual data movement is from the chunk server to the client. Figure 8.5 illustrates the various steps involved.

Failures: Failures of any of the involved entities in the entire operation can be handled by a suitable retry. The client can cache the information about the location of the chunkservers containing the chunks corresponding to the file. This allows the client to use the cached information to read directly from the chunkserver instead of asking the Master for the location of the chunkservers. The client can use its own cache policy as to how long the location information is kept in the cache.

An additional optimization that is possible is for the server to send the location of chunkservers of chunks of the file beyond the requested chunk. This optimistic buffering reduces network traffic and is suitable for applications that read files sequentially.

- **Write:** The implementation for the write operation is a bit more involved than the read operation. This is due to the fact that a read

operation, and even a concurrent read operation, does not modify the state of the file. A write changes the contents of the file and hence the system should make clear the guarantees it provides on a write operation, in particular a concurrent write.

The client initiates a write operation by specifying the file name and the offset where it intends to write the data. The master server reponds with the location of the primary and the secondary chunk servers that store the required chunk. Assume for now that the chunk has enough space from the specified offset to hold the data to be written.

There are two steps in the operation: (i) sending the data to *all* the chunk servers, and (ii) ensuring correctness of the operation in coordination with the primary chunkserver. The client can perform the first of these steps by sending the data to all the chunk servers in any order it chooses to. This step completes when all the chunk servers acknowledge the client about receiving the data. The chunk servers keep the data in their local buffers at this point.

For the second step, the client approaches the primary server with a write request so that the primary chunk server can assign a serial number to the write operation. These serial numbers help in supporting the necessary serialization in case multiple clients attempt to write to the file simultaneously at overlapping locations. The primary sends the write request to all the secondories along with the serial number. The secondaries have to apply the write updates in the same serial order specified by the primary. (See also Question xx)

Figure 8.6 shows the steps involved in the write operation. The secondaries report to the primary about the success or failre of the write operation. The primary accordingly informs the client of the success or failure. If any secondary enounters an error in the write operation, then the primary responds to the client that the write operation has failed. Notice that the write might have succeeded at the primary and some secondaries. So, if some secondaries report a failure to complete the write operation, then the corresponding region of the file stays in an inconsistent state. Applications are expected to handle such errors and inconsistencies. If the client gets an error from the primary for the write operation, the client can choose to retry.

For writes that straddle a chunk boundary, the operation is split into multiple write operations. Each such operation is treated as a separate write and hence could be interleaved with write operations from other

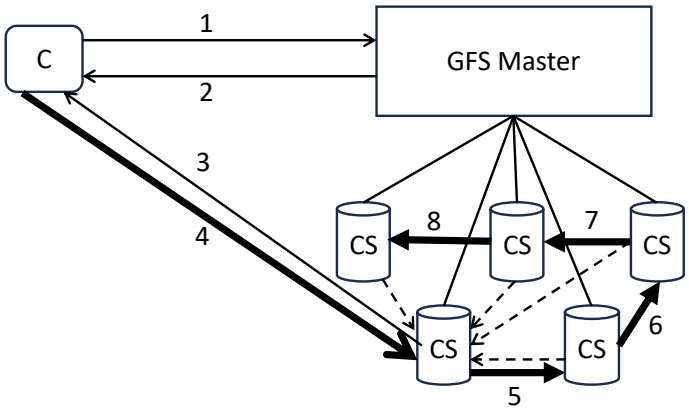


Figure 8.6: Figure illustrating the steps in a write operation in GFS. The arrows indicate the direction of the data transfer. The arrows with dashed lines indicate acknowledgements from chunk servers to the primary chunk server. These acknowledgements may arrive in any order and hence numbers are not provided.

clients with different serial numbers. Therefore, the contents of the file may be a mixture of writes from different clients. However, as long as all the operations succeed, the order of data in all the replicas is exactly identical since these are written in the serial order assigned by the primary.

Failures: Notice that it is likely that the primary or a subset of the secondary chunk servers may fail to complete the write operation. In this case, the client can retry the operation. However, on subsets of chunk servers that the operation succeeds, the modified region is left in an inconsistent state.

- **Record-Append:** This is an operation that is driven by the nature of workloads. In this operation, the client provides only the data to be written, and the file system adds the data at an offset chosen by the file system. The file system guarantees however that the data will be written at least once atomically.

  Just like the write operation, here too, the client gets the location of the primary and the secondary chunk servers, holding the last chunk of the file, from the master server. The client then sends the data to be written to all the chunk servers. Assume for a moment that the data to be appended fits within the last chunk. In this case, the

primary adds the data to the end of the chunk, informs the secondary chunk servers about the offset to use for this append, collects stauts of the operation from the secondaries, and then informs the client of the status.

In the case that the space in the current chunk is not enough to accommodate the append, the primary server takes the following steps: pad the space in the current chunk, inform the secondaries to do so, and inform the client that the append operation failed due to lack of space. The client can then retry the operation.

Failures: Notice that if the record-append operation fails at one or more of the replicas of the chunk, then the client can retry the record-append operation. This means that the replicas of a chunk may contain different data. However, the record-append operation is successful if all the secondaries report a success on the operation. Atomicity is guranteed due to the primary choosing the offset of the append. Hence, GFS supports the append operation with *at least once atomically* semanitics.

It is up to the applications to handle the case of duplicate records and padded data. The former is usually handled with having unique identifers for each record. The latter is easily handled with techniques such as checksums which can flag padded data.

- **Other Operations:** GFS supports other operations such as a `snapshot`. In this operation, the goal is to make a copy of a file or a directory. GFS does this in a way that this copy is instantaneous irrespective of the size of the file (or the directory). This operation is useful in cases where a user wants to create a checkpoint or a copy of a file.

  The snapshot operation is instantaneous since GFS adopts a copy-on-write approach. This copy-on-write approach is a standard systems practice that saves the expense of writing out the copy when the resource does not undergo any changes subsequently [4].

  To perform a snapshot operation on a file f, the GFS Master first invalidates/revokes all the leases on all the chunks that correspond to the file f. This means that any future write operation to f will need to go through the Master server. At this point, the Master can

[4] Most Unix style operating systems use this copy-on-write practice when a child process is created. The child process shares all the pages of the parent process and the copy-on-write flag is set on all these pages [10].

make a copy of the chunk by creating a new chunk to enfore the copy-on-write semantics. Notice that all the chunkservers that hold the chunk will also need to make a copy. These copies are created on the same chunkserver where the original chunk resides so as to reduce the network traffic involved in copying the data. The Master uses reference counts to know if a file is part of a snapshot request. A reference count bigger than 1 indicates that the file is part of a snapshot request.

8.5.2 Leases

For every chunk, GFS assings the role of a primary chunk server to one of the three chunk servers that store a copy of the chunk. This role of a primary chunk server is assigned as a lease for a duration of 60 seconds. The currently designated primary chunk server can get extensions of the lease indefinitely. The extension request and grant is sent along with the periodic heart-beat messages that go between the master server and the chunk servers. The master can revoke the current lease for various reasons including instances when the file is being renamed and the master wants to prevent changes to the file contents, the master loses communication with the primary server due to any failures, and so on.

8.5.3 Chunk Size

GFS set the size of each chunk to be 64 MB. This choice is justified given that the system expects to store very large files. In such cases, the large chunk-size helps in several ways. For instance, the client need not interact with the server until the operations (read/write/record-append) cross a chunk boundary. All of these operations can be performed based on the identifier of the primary chunkserver that the client obtains from the Master server during an earlier request to the same chunk. There is also the possibility that the client can keep a persistent network connection to the chunkserver across multiple operations directed at the same chunk and the chunkserver, thereby optimizing on network traffic. In addition, a large chunksize also reduces the size of the metadata information that the Master has to keep.

There are some usual disadvantages of using a large chunksize. One potential issue is internal fragmentation where the filesize is much smaller than the chunk size. GFS handles this by actually storing chunks as regular files in the underlying operating system. The space alloted to the chunk is increased on demand. GFS calls this as *lazy space allocation*. Lazy space allocation however does not handle one potential problem, hotspots. If a

file contains only one chunk, and many clients want to read from this file, all these requests will be directed to the primary chunkserver storing the corresponding chunk. GFS handles this issue by other means such as a combination of increasing the replication factor, spreading read operations to not just the primary chunkserver but also the replicas/clients, stagger the requests, and the like.

8.5.4 The GFS Master

We now provide more details of the GFS master server. The master server keeps the metadata corresponding to the files and the chunk servers that hold the chunks of the files in its primary (volatile) memory. This mapping is not stored in any permanent storage. The information on which chunk server is the primary chunk server for a given chunk is kept fresh by constant polling of the chunk servers. Doing so also takes care of master server handling the failures of the chunk servers. Chunk servers that do not respond to the poll messages are assumed to be failed and the responsibility is quickly reassigned to a secondary chunk server.

The master server also maintains a log of the operations that result in changes to metadata. This log is kept on persistent storage, provided with checkpointing so as to minimize time for recovery, and also serves to define the order in which concurrent operations are affected. Response to client operations is sent out only after the corresponding log record is flushed to permanent storage.

Unlike Unix style file systems, GFS does not support per-directory data structure of the list of files in that directory. File names are stored as a lookup table that maps the full pathname of the file to its metadata.

For operations by the master server that require changes to the namespace, the master server uses locks over appropriate regions of the namespace. This avoids the situation that time consuming operations at the master lead to unavailabilty in servicing of any file.

8.5.5 Fault Tolerance

GFS is designed for a scale where there are several commodity processors acting as chunk servers or as a master server. At this scale, components failures are the norm. A detailed study by Pinheiro et al. [99] shows that disk drives experience an baseline Annualized Failure Rate (AFR) ranging from 1.7% to 8.6% over the first year of their lifetime to their fifth year of lifetime. Hence, the system must take steps to ensure operations continue

even in the presence of intermittent failures. GFS takes the following steps to address this aspect.

- Replicating the Master State: Just as a chunk server can fail, the master server that is also a commodity server can fail. This situation can render the whole file system unavailable. To prevent such an occurrence, GFS replicates the state of the master. The state of the master includes metadata about the file namespace, information on chunk servers, and the mapping between files and chunk servers. In addition to replicating the state of the master, GFS also keeps the file namespace and the chunk namespace in persistent memory by logging modifications to the two namespaces into persistent storage on the master's disk and also on to the disk at other remote machines. A change to state is said to be committed only after such a change of state is recorded to a persistent storage.

  On the failure of a master, the replicated log is useful in quickly creating a new master process. To keep this change transparent to the user, clients use only an indirect way of addressing the master. For instance, the indirect address of the GFS master could be a DNS alias instead of an IP address.

- Replcation of Chunks: Each chunk is replicated three times by default. Users can increase the replication factor based on their application requirements. Other mechanims such as erasure codes could be tried instead of full replication.

- Data Integrity: Each chunk is treated as a logical collection of 64 KB blocks. Each such 64 KB block is provided with a checksum so that the chunk server can verify each block with the checksum before returning data to any request. The checksums are kept in memory and also stored in persistent storage with logging.

  If there is a failure to match the checksum, the chunk server can declare the chunk to be in error to the master and report a failure to the request. This allows the master to create a new replica of the chunk from other chunk servers holding the chunk and inform the chunkserver that has the wrong chunk to delete the chunk.

  Having a large chunk size reduces the overhead of computing the checksum on every read request. The verification of the check sum can happen in parallel to the read. As GFS is geared more towards Record

Append instead of write, checksum can be updated incrementally during the Record Append operation. This choice also means that a regular write is slow compared to the Record Append since a regular write requires recomputing the checksum.

Chunk servers also use their idle time to verify the checksums of the chunks they hold so as to identify any *bad* chunks. By reporting to the master, these bad chunks can be removed from the system by replicating the bad chunks with good chunks.

- Shadowing : GFS maintains *shadow master* shadow master processes that tails the progress of the master up to a small delay. The shadow processes can allow read access to files in case of unavailabilty of the master server. Shadow master processes update their state frequently based on the logs written by the master server and apply the updates from the log in the same order to their local data structures. Notice however that the content served by shadow processes can be stale.

8.5.6 Design Features and GFS

Let us now understand how GFS fares with respect to the common design features listed in Section 8.1.

To ensure data durability, GFS replicates each chunk of the file at three different chunk servers. The degree of replication can be increased. Notice from the GFS operational summary that the replicas are held consistent to a great extent. (Question: When can replicas diverge?) This ensures that the file contents can be recovered when a chunkserver crashes. Even the state of the GFS Master is replicated so as to ensure that the file system state can be quickly recreated in the event of a crash of the Master server.

GFS maintians availability by having mechansims such as (i) replication of chunks at multiple chunkservers, and (ii) a shadow master to keep the system available in the event of crashes to the master server. The presence of a single master may mean a single-point of failure, mechanisms such as shadowing and state replication allow for increased availability and fault-tolerance and recovery.

GFS ensures that concurrent writes and record-appends come with guarantees. The write operation is serialized by the primary chunk server by providing and enforcing the update order. This ensures that all clients see the data in the same order. The primary chunk server assigns athe offset for each record-append operation and all replica chunks are updated by using

the assigned offset. This ensures that each record-append is performed at least once atomically.

To ensure scalability, GFS allows for having multiple chunk servers store the data.

GFS makes the choice of using a flat namespace instead of an hierarchical namespace that is more common to Unix type of file systems. The namespace and metadata that GFS maintains is more lightweight and this allows for keeping the metadata in volatile storage of the GFS Master allowing for quicker access to the metadata.

8.5.7 Summary

The design of GFS is geared towards a system that works well with batch jobs. Characteristics of batch jobs such as having a long job time, tolerance to short failures, ability to view the computation as record processing, make it suitable for systems such as GFS.

GFS coupled with Map-Reduce allowed Google to support a strong search engine and other Google products. As the nature of products and services moved to more real-time, the deficiencies of GFS such as having a single point of failure at the master server began to make an impact.

8.6 Colossus

With the increase in the amount of data that Google deals with and also with the increase in the vareity of services with different requirements, a file system that can scale better than GFS is seen to be desriable. This led to the development of Colossus [61], the enhanced version of GFS. Colossus is a cluster-level file system and is one of the main building blocks of supporting both Cloud and Google products.

The Colossus file system is designed to scaled for beyond Exascale data while at the same time offering high availability and being able to support specific needs of multiple applications. The goals of Colossus is to arrive at more predictable tail latency beyong being faster and bigger. Some of the functionality of Colossus is made possible due to the distributed metadata model that is uses. The main components of the Colossus are (i) the client library, (ii) the Colossus Control Plane, (iii) the Metadata database, (iv) the data servers, and (v) the Custodians. Figure 8.7 shows the schematic architecture of Colossus. In the following, we elaborate on each of the above components.

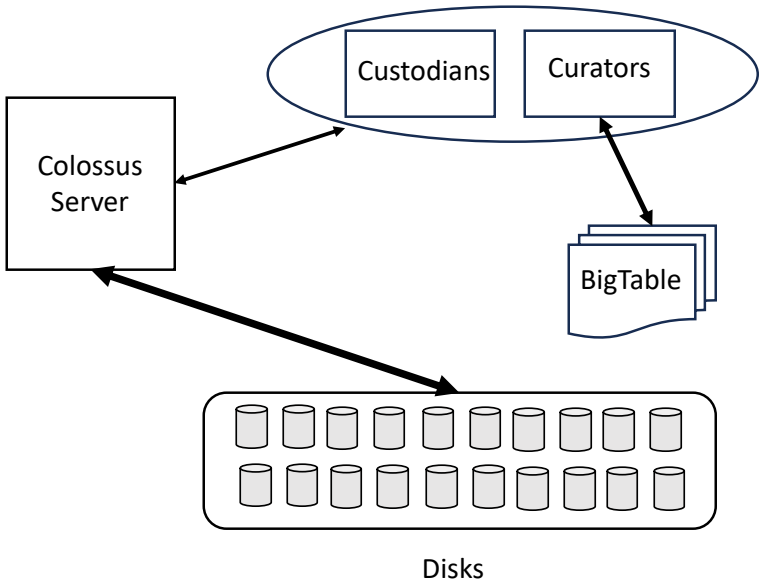


Figure 8.7: Figure illustrating the schematic architecture of the Colossus file system.

The Client Library

The client library packs a lot of functionality and applications use the client library as their primary interface to use Colossus. The client library has support for a variety of functionality that applications can tailor based on their requirements and their cost-performance tradeoffs.

The Colossus Control Plane

The consists of Colossus Curators which keep the file system metadata. The number of curators in the system can scale horizontally. Clients interface with the curators for file control operations such as file creation.

The Metadata Database

Curators keep the metadata in a Google No-SQL style database, Big Table [24]. (See also Chapter 9.4 of this book.) This is one marked departure from GFS. Using only main memory for metadata as done by GFS is not scalable. On the other hand, moving the metadata to a database helps scale the file system to a factor of 100 or more.

The Data Servers

Colossus system is built to work with a variety of data servers. Notice that with horizontal scaling, new disks are added to the system and old disks are retired. The characteristics of the disks that are in use at any given time vary in their size and access models too. In Colossus, the disks are attached to the network so as to minimize the number of hops that data stays on the network.

The Custodians

The scale of operation of Colossus suggests that periodic background services are needed to keep the system up and running. These background mangers are called as Custodians and support operations for maintaining the durability and availability of data, overall system efficiency, disk space balancing, and RAID reconstruction.

8.6.1 Overall System

Colossus ensures that data that is likely to be more frequently accessed, hot data, is kept in flash drives. This ensures that hot data is served with less latency. Other data is moved to disk storage. Further, newly written data is spread evenly across the drives in a cluste. Periodic data rebalancing also ensures that data is moved to newer and larger disks eventually. All these tasks are possible since Colossus aims to take benefit of the fact that across applications, the data access patterns vary significantly.

The scale at which Colossus operates essentially means that it uses a large amount of commodity hardware. In practice this comes with failures to components are common. Any time instant, there will be some components that have failed. To address fault tolerance, Colossus custodians undertake fast background recovery and keep failures transparent to the user and provides highly durable and available storage.

8.6.2 Other Considerations

Given the deluge of data anda the nature of its access, two phrases that are popular today in the systems community are hot data and cold data. Hot data refers to the data that is being accessed more frequently presently, whereas cold data refers to data that is not frequqently accessed. This classificaiton is important when one looks at applications such as photos being accessed online, e.g. Google Photos, or Facebook multimedia posts,

where such items are accessed very frequently aroud the time they are made available (posted), and over time are accessed very rarely.

The classfiication of data into hold and cold data is important from a systems perspective in the following way. In a file system with multiple disks, it is essential to keep both kinds of data on all disks. Otherwise, if a disk largely has cold data, then the usage of that disk is disproportionately low while a disk with largely hot data get used more often potentially leading to faster failures. Colossus, therefore, aims to keep a percentage of each disk with hot data and the rest of the disk with cold data. To ensure this condition over time, Colossus rebalances disk contents and distributes new data evenly across available disks.

In addition, a subset of the data, the most frequently being accessed, is put on flash drives to minimize access latency.

8.6.3 Design Features and Colossus

Colossus is prepared with scale in mind. The system allows for the addition of disks to enhance the storage capacity of the system. Current usage levels already exceed the exabyte mark [**?**].

In a departure from GFS, Colossus brought down the chunksize from 64 MB to 1 MB. The design of Colossus is towards having a large number of distributed storage nodes, each storing a large number of small files. As part of this restructuring, notice that the size of the metadata grows compared to that of GFS and the expected number of files that Colossus stores. To address this issue, the metadata is no longer stored in the volatile memory of the Master, but in a database, Bigtable.

Having a distributed master node design also improves the availability of Colossus and makes the system resilient to the single point of failure that GFS has to contend with.

Colossus provides for data durability by using replication.

8.7 Systems for Object Storage – Haystack

Apart from Google, Facebook is another platform that deals with massive amount of data. Files handled by facebook include audio/video files and photographs as the platform is used primarily to share photos and audio/video content across friends. So, the file system should support quick delivery of content at a planet scale. While Content Distribution Networks (CDNs) also offer a solution that appears similar to that of serving files across the planet, CDNs are typically optimized for frequently used data

items. On the other hand, the kind of workloads that Facebook deals with will have a long tail and for overall user experience, it is important to also see how the latency of requests corresponding to the long tail can also be minimized. In this case, CDNs may not offer a good solution since CDNs mostly rely on caching mechanisms and objects in the long tail do not make it to the cache by definition.

In 2010, Facebook rolled out its distributed object store called HayStack [17] that is primarily targeted at storing photos efficiently. We now describe the shortfalls of existing systems and describe the mechanisms that Haystack uses to store and serve photos efficiently.

8.7.1 Shortfalls

One of the natural ways to cater to the storage and serving of photos would be to store each photo as a separate file in a file system and use NFS to scale to the large number of photos that are shared on the platform. However, this comes with multiple drawbacks summarized in the following.

In a traditional file system, each file is stored with a large amount of metadata including access control. The typical Unix file system creates access control to the owner, the group(s) the owner belongs to, and others. But, a system that has a large number of read operations (viewing photos) and very few writes does not require elaborate mechanisms for keeping file access control information. Typically, only the creator of the file is the owner and other users can only read the file and will not be able to write to the file. Such unused fields, especially at billion scale files, bloat the metadata that will be read everytime the corresponding file is accessed increasing the bottlenecks and reducing the throughput.

As mentioned previously, CDNs do a good job of storing and serving objects that are accessed most frequently. With systems such as Facebook also getting significant volumes of requests to stale/infrequent data, CDNs do not cater to such traffic thereby increasing the latency on such requests. Thus, relying on CDNs alone which using principles from caching may not cover all possible usecases.

For higher throughput, it is an advantage to keep as much metadata in memory as possible. However, if the metadata corresponding to each file is large, this limits the amount of metadata that can be stored in the main memory. So, it is important to limit the size of the metadata and see which fileds of a traditional file system metadata are expendible for the chosen workloads.

8.7.2 The Haystack System

The Haystack system [17] has three main components: the Haystack store, the Haystack Directory, and the Haystack Cache. The Haystack Store has the filesystem metadata and encapsulates the persistent storage system for photos. The Haystack Directory maintains the logical to physical mapping along with application metadata. The Haystack Cache functions as an internal CDN and serves requests for hot photos – these requests need not be served from the Haystack Store. The Haystack Store sits on top of an existing file system. Facebook used XFS as XFS has certain advantages to the considered workload. In the following, we will detail these components.

The Haystack Store

The Haystack Store is arranged via physical volumes. Each physical volume holds millions of photos. Haystack stores multiple photos in a single file and hence the files are are very large. Physical volumes across machines are grouped into logical volumes. Haystack replicates a photo on a logical volume by writing the contents of the photo is written to all the physical volumes that constitute the logical volume. This replication allows Haystack to address fault tolerance due to failures of hard drives, disk controllers and the like.

The Haystack Store machine accesses a photo using only the id of the corresponding logical volume and the file offset at which the photo resides in the logical volume. A HayStore machine represents a physical volume as one large file. The file has one superblock followed by a sequence of needles. Each needle represents a photo stored in Haystack.

Retrieving needles quickly is an essential ability of the Haystack Store. To this end, each Haystack Store machine keeps open file descriptors for each physical volume that it manages and also an in-memory mapping of photo ids to the filesystem metadata (i.e., file, offset and size in bytes) that is useful in retrieving that photo. Each machine also maintains an in-memory data structure for each of its volumes. That data structure maps pairs of (key, alternate key) to the corresponding needle's flags, size, and offset.

The Haystack Directory

The Haystack directory has four main functions as listed below.

- The directory provides a mapping from the logical volumes to the physical volumes. This mapping is useful for web servers to construct

the URLs for photos being requested and is also useful in processing an upload.

- The directory also balances the load across the logical volumes and spread reads across the physical volumes.

- The Directory determines whether a CDN or a Cache handles the request to access a given photo.

- The Directory marks certain physical volumes as read-only once these physical volumes reach their full capacity or becuase of operational reasons.

Typically, over time, the system will add more physical volumes as the existing volumes reach thier storage limits. The new volumes are marked for write. Only volumes that are marked for write can receive uploads.

The Haystack Cache

The cache keeps a mapping of the photo id to the location of hte photo. The cache is essentially a distributed hash table. Haystack uses two rules to determine if an object/photo is kept in the cache: (i) the request being served from the store comes directly from a user and not a CDN, and (ii) the photo is fetched from a machine that is enabled to receive uploads. The second of the two conditions indicates that most recent photos are the ones that are recently uploaded and hence are being written to a write enabled macine.

File System

Haystack can be viewed as a distributed object store that sits on top any generic file system. However, Facebook used XFS as the underlying file stystem. First, the blockmaps for several contiguous large files can be small enough to be stored in main memory. Second, XFS provides efficient file preallocation, mitigating fragmentation and reining in how large block maps can grow. Using XFS, Haystack can eliminate disk operations for retrieving filesystem metadata when reading a photo. However, it does not imply that Haystack can guarantee every photo read will incur exactly one disk operation.

8.7.3 Operations

In this section, we discuss the steps that Haystack uses to store a photo, serve a photo, delete a photo, and so on.

Storing/Uploading a Photo

When uploading a photo into Haystack web servers provide attributes such as the logical volume id, key, alternate key, cookie, and data to the Haystack Store machines. Each machine then synchronously appends needle images to its physical volume files and updates in-memory mappings as needed.

Operations on photos such as rotations have to create a new needle. This new needle will have the same key and alternate key as the original photo. There are now two possibilities. The new needle is stored in the same logical volume. In this case, the new needle is appended to the same physical volumes as the original needle. The Haystack store uses the rule that the needle with a larger offset is considered as a later version. If the new needle is stored in a different logical volume, then the Haystack Directory updates the metadata entries corresponding to the original needle so that all future requests fetch the new version.

Serving a Photo

Consider a user requesting for a photo thorugh a browser. This request goes to the web server and can be served by the external CDN of the internal Haystack Cache if the request missed the CDN. To facilitate a smooth hand-off between the CDN and the Cache, the web server constructs a URL for the request as follows. Each URL is of the form `http://<CDN-Name>/<Cache-Name>/Machine-Id/<Logical Volume, Photo>`. In this encoding, notice that the first part of the URL specifies the CDN that can possibly serve the request. If the request is not satisfied at the specified CDN, the CDN can remove the corresponding part of the URL and forward the request to the Haystack Cache specified in the URL. If the Cache fails to serve the reuqest, then the Cache can remove the corresponding part of the URL and forward the request to the machine (Haystack Store) that can serve the request. At this level, if the photo is found in the specified volume at the given offset, it is served. Requests that go directly to the Haystack Cache follow a similar process except that the URL is missing the CDN specific information. Figure 8.8 shows the interactions between the user, web server, the CDN, and the Haystack for serving a request.

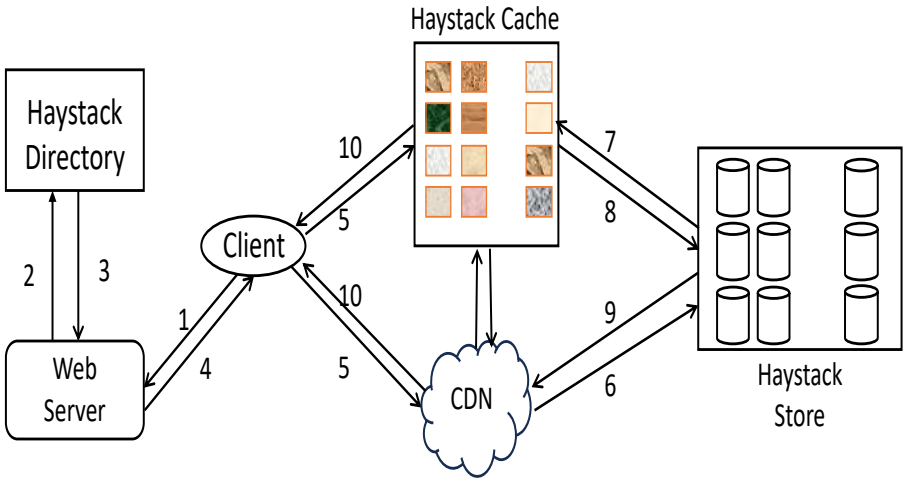


Figure 8.8: Steps involved in reading a photo via haystack.

The process of serving the photo from the haystack Store follows the steps below. The request to the Store is accompanied with the logical volume id, alternate key, and the cookie fields. Recall that these attributes are part of the needle data fields. The cookie field, being a random number assigned at the time of upload and is embedded in the URL, prevents guessing attacks that can prepare valid URLs to mount denial-of-service attacks. The Haystack Store reads the volume from the provided offset, makes sure that the photo is not (marked) deleted, verifies the cookie, and then passes the photo to the Haystack Cache.

Delete a Photo

Deleting a photo is a logical operation. The Haystack Store machine that has the photo sets the delete flag in both the in-memory mapping and synchronously in the volume file. Since requests to retrive photos first check the in-memory flag, such photos are never returned as the answer to the request.

As with any logical delete models, the space held by deleted photos has to be reclaimed via a future internal operation. This is called compaction and reclaims space held by deleted or duplicate needles. Periodically, each Haystack Store machine goes through the entire set of needles, copies the valid needles to a new file, and at the end of this process sets the in-memory structures to the new file.

8.7.4 Optimizations

Haystack includes the following optimizations to improve performance.

- **Batch upload:** Notice that disks are good at batch updates since positioning the disk for access is a significant cost. So, Haystack prefers to batch uploads to the disks as much as possible. From a workload perspective too, most users upload a set of photos than a single photo. This enables Haystack to use batch updates in an album together.

- **Memory Footprint Minimization:** Instead of setting flag field on for deleted photos, Haystack opts to write a 0 in the offset for such photos. Further, Haystack Store machines do not store the cookie field in memory and checking the cookie is done based on the needle read from the disk.

- **Index File:** This is a (re)boot time optimization used by Haystack. When a Haystack Store starts up, it can read the entire set of physical volumes and build the required mappings to be held in-memory. This will be slow and time-consuming as this involves reading Terabytes of data from disks. The index file is akin to a checkpoint of the last known in-memory mappings. Restarting using this checkpoint version of the can result in some inconsistencies especially if the index file corresponds to stale information. These inconsistencies can create orphan needles, needles that do not have corresponding index records. To cater to this situation, the Haystack Store machine maps orphans to valid index records. If an index record corresponds to a deleted photo, the Store machine relies on the usual procedure of reading the entire needle for a photo and checking the delete flag on the needle. If the flag is set, then the index record is updated in the in-memory mapping and also notified to the Haystack Cache. REDO

8.7.5 Fault Tolerance

Systems such as Haystack by virute of running on multiple commodity machines are prone to failures of various kinds inlucding faulty hard drives, erroneous RAID controllers, bad motherboards, and so on. However, the scale of the systems is such that availability is key. To address this situation, Haystack has two techniques for detection and repair.

Fault detection is usually done by background processes, called *pitchfork* in Haystack. These background processes tests the health of every

component and flags the faulty ones for follow-up action. This is only a diagnostic measure. The real repair has to be done offline.

8.7.6 Design Features and Haystack

We now summarize the design features of Haystack in the context of the features listed in Section 8.1. One of the distinguishing features of Haystack is that its design is more oriented towards minimizing, or eliminating, the size of metadata and the number of disk accesses needed to serve a photo request. This motivates the design of Haystack to keep metadata so small that the entire metadata fits in the main memory.

Haystack keeps photos in its store and interfaces with the store more as an object store with each on disk XFS file storing multiple photos. Haystack keeps the files large and relies on XFS to use a large block size.

Given that Haystack is built to serve a large user base, durability and availability concerns are paramount. For purposes of durability, Haystack relies on replication. Each photo is replicated in multiple geographical locations and also at all physical volumes corresponding to a logical volume. Recall that a collection of physical volumes on different Store machines is grouped as a logical volume. This redundancy allows for data durability and protects against loss of data due to failures of the hard drives, hard drive controllers, RAID errors, and so on. Access to Haystack is via an embedded URL that directs the browser to a CDN or a Haystack Cache. This model ensures high availability with simple mechanisms.

In the typical usage model of Haystack, updates to photos are the only way to modify the contents of the objects. These updates can be done only by the owner or creator of the object. These updates are treated as creating new needles for referring to the new content and the most recent needle for a given photo has the largest offset. So, Haystack does not place much of design emphasis on concurrent operations.

Haystack addresses the scale issue by allowing for machines to be added to its Store. In fact, Haystack marks all full volumes to be read-only while new volumes deal with writes. This subtle dinstinction allows Haystack to optimize for read-only and write-only operations to volumes and also lays out a cache policy that is tailored to fresh data.

8.8 The Facebook Tectonic Distributed Storage System

Haystack [17] is used for storing blobs and supports one kind of applications that Facebooks deals with. As Facebook had to deal with multiple such applications (aka tenants), each with its own distributed storage and file system, the ensuing collection of small specialized storage systems resulted in inefficiencies including the development, optimization, and maintenance of multiple systems. For instance, tenants based on data analytics at warehouse scale data were using HDFS and are optimized for batch processing and hence aiming for better throughput over latency. However, HDFS clusters are limited in size and results in managing tens of HDFC clusters per datacenter just to store the data reuqired by the data analytic jobs. Blob storage that is spread across Haystack and f4 for hot and warm blobs, respectively, resulted in poor resource utilization.

To address this issue, Facebook preapred the Tectonic distributed storage system [94] that can scale up to exabyte storage and allow for multiple tenants while making it possible to support isolation across tenants and optimizations specific to each tenant. Tectonic is arranged as a set of clusters that are local to a datacenter and provides a fault-tolerant storage model. Applications/tenants can also choose to georeplicate an entire cluster for better reliability.

A tectonic cluster has three main parts: a chunk store, a metadata store, and a stateless metadata services system. The chunk store is a collection of storage devices (hard drives) that store and access data chunks. The metadata store keeps metadata related to the files and chunks, and the metadata services component that run in the background to maintain consistency across metadata layers in addition to others.

8.8.1 The Chunk Store

The Chunk Store is essentially a collection of disk drives. For scaling, any number of disk drives can be added to the chunk store. In addition, the chunk store is oblivious to the actual higher-level abstractions such as files and blocks. This separation between how the storage nodes work and how the files and blocks are handled by the file system client is an important aspect in providing seamless multi-tenant support.

Each storage node can have multiple disk drives and each such node implements XFS [57]. Each storage node is provided with a large solid state drive for keeping XFS metadata and caching hot chunks. Keeping XFS

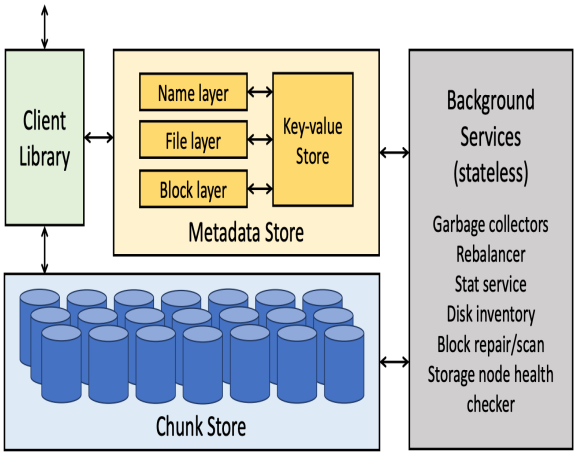


Figure 8.9: The architecture of tectonic file system. Figure taken from [94].

metadata on SSD storage is useful for applications that use blob storage since most blobs are stored wirtten as appends that only affect the chunk size.

Tectonic supports each block to be stored as a set of N replicated chunks or to be stored as a Reed-Solomon encoding where the block is replicated as N chunks and K parity chunks. This choice can be made by the tenant based on its requirements. This flexibility of tectonic to allow for tenant-specific optimizations is crucial for supporting a unified system that can cater to the needs of multiple tenants.

8.8.2 The Metadata Store

The metadata store keeps key-value stores over three components: (i) name layer, (ii) file layer, and (iii) block layer, and the metadata store is sharded across these three layers. Each of these key-value stores is implemented as stateless microservices using ZippyDB [92].

The key-value store are stored on metadata nodes in SSD and are also sharded with replication at the shard level. The nature of sharding for the various key-value stores is shown in Table 8.1. Shards are replicated with Paxos [80] for fault tolerance. Each metadata node can host several shards. This allows shards to be redistributed in parallel to new metadata nodes to react to failures and reduce recovery time. Any replica can serve reads to the key-value store and reads that are required to protect strong consistency

Layer	Name	File	Block
Key	(dir_id, subdir_name) (dir_id, filename)	(file_id, block_id)	block_id (disk_id, block'id)
Value	subdir_info, sudir_id (file_info, file_id)	block_info	list<disk_id> chunk_info
Sharded By	dir_id dir_id	file_id	block_id block_id
Mapping of	dir to subdirectories dir to list of files	file to list of blocks	blocks to chunks disk to list of blocks

Table 8.1: Metadata layers

are served by the primary.

The Name layer maps directories to the files and sub-directories they contain. The namespace is flat or expanded. For instance, if a directory `test` contains two files `villa` and `tower`, then there are two keys (`test`, `villa`) and (`test`, `tower`). The file layer maps files to the set of blocks corresponding to the file. A file is an ordered list of blocks where blocks serve as an abstraction for a contiguous string of bytes.

The key-value store does not provide cross-shard transactions. This limits certain filesystem metadata operations and complicates some multi-step operations. For instance, consider renaming a file `test` in a directory as file `new` even as the the file `test` and the new file will be on different shards. If this rename overlaps with the creation of a new file with the same name in the same directory, then care should be taken so as not to leave the file system in an inconsistent state. A possible inconsistency can arise due to the following play of events. For renaming the file, the three steps needed are to get the fileid for the file `test`, add the file named `new` as a new file, and map the file name `new` with the file id of the file `test`. To create a file with name `new`, the steps involved are to credate the new file ID, and map the file id to the filename. If the above operations of renaming a file and creating a file interleave such that the steps of creating the file are executed after the first step of the rename operation and before the second and third steps of the rename operation. In this case, the third step of the rename operation can erase the mapping introduced by the file creation.

One of the issues with applications that use blob storage is the possibility of hotspots in searching within the directory structure. The layered metadata approach of Tectonic's aims to avoid such hotspots in directories and other layers by separating searching and listing directory contents from

the name layer and reading file data through the file and block layers.

8.8.3 The Metadata Services Component

The scale of the Tectonic system and its usage implies that repairng inconsistencies/loss of data, garbage collection, and health monitoring of devices are essential. For instance, an action that is left incomplete due to an intervening client failure can lead to inconsistencies in the metadata. Similarly, performing lazy deletion, which is usually done to reduce latency in real-time operations, requires a later phase of cleaning up. Storage capacity limits, load imbalances are also possible due to the scale of the operations.

The Metadata services component performs several services in the background to address the above situations. Some of the services include rebalancer, garbage collector, statistics, storage node health monitoring, disk inventory, and block repair/scan.

Copysets is another techinque that tectonic uses to improve fault tolerance and data availability even in the presence of coordinated disk failures. When using replication, there is a tradeoff between the likelihood of data loss and how much data is lost per failure event. For instance, under a three way replication with uniform failure rates, the probability that a particular block is lost is increases as any set of three failures will result in the loss of a block that is stored on these three devices only. However, the amount of data lost will be low since it is also highly likely that some replicated device will have the data that is stored on only one or two of the failed devices.

To decrease the probabilty of data loss, the idea of copysets is to create independent subsets of the devices. Each such independent subset is called as a copyset. Each block is then replicated in the devices in a chosen copyset. In this case, the event that results in a data loss corresponds to the event where all (or a subset of) the devices in the copyset fail. On the downside, using copysets however increases the amount of data lost per failure event. Another downside of using copysets is that repair traffic is now all directed to the devices in the same copyset. This skews the repair traffic too. There are a few ways to deal with this skew [27].

8.8.4 The Tectonic Client

For the tectonic client, the unit of abstraction for read/write is a chunk. The Client Library interacts with the Metadata Store to locate chunks, and updates the Metadata Store for filesystem operations. Tectonic supports only single-writer semantics. The exact support for write operations is discussed

in the next section. If a tenant needs support for multiple/concurrent write, should build their own serialization semantics on top of the support offered by tectonic.

8.8.5 Operations and Semantics

Tectonic supports single-write, append-only semantics. Thus, tectonic does not support concurrent writes to a file. All writes, even by the single writer, also are appends. On opening a file, clients are given a token the information corresponding to which is stored with the file in the metadata layer. For performing a write operation, the client has to mention the token along with the data. Only the most recent write token is allowed to mutate the file metadata and write to its chunks.

A write (append) is deemed to be complete if a majority of all chunk replicas of a block in order to commit the block are written durably. This is a quick optimization to reduce the tail latency for the write operation [33]. For full block writes, tectonic adopts another technique to know which of the chunk replicas are written first. With unpredictable delays, it is not possible for a client to know what are the best replicas to write to. So, tectonic uses a policy called as reservation requests . In this solution, the client sends small ping style requests to multiple storage nodes. The ones that respond quickly to the client request are used to write as part of the majority. This technique reduces the tail latency.

For partial/small writes (appends), the model used is slightly different. In this case too, the write is deemed to be complete once a majority of the nodes in the replica perform the write operation. This can lead to a problem if multiple concurrent partial writes append their data on different replicas. This results in inconsistencies even if the metadata seems correct since the order in which the replicas are written to can be different across the clients.

To avoid this problem, tectonic enforces two rules. Firstly, only the client that created a block is allowed to append to that block. Secondly, before deeming the operation complete, update to the metadata including the new block length and the checksum has to be finished.

The above mechanisms mean that applications can expect read-after-write consistency. This follows since the block-length and checksum after each quorum append are commited to durable storage and metata before acknowledging to the application.

8.8.6 Design Features and Tectonic

Tectonic is designed to be scalable and scales to Exabyte scale storage levels due to its decentralized nature and ability to add more hardware resources as needed. We notice that except for NFS, the ones covered in the chapter with respect to distributed fle systems or distributed object stores do address the scalability issue.

Data consistency in Tectonic is ensured since the only operation that changes the contents is via the append-only operation that can be done by the data owner. Applications, also called as tenants in Tectonic, can provide for more diverse write support, such as the ability for multiple users to write and enforce consistency guarnatees. To ensure consistency of the metadata, Tectonic employs periodic background checks.

Tectonic provides data durability by using replication. Each data block is replicated close to three times – the authors of tectonic mention that this is a small improvement over the replication factor of haystack. Beyond the system default, replciation Replication for durability, geo-replication

Availability:

Namespace/Metadata To support a wide variety of applications and to provide for more scalable design, tectonic uses a nearly flat two level namespace. Tectonic supports the notion of a directory and a file. However, the contents of directory are stored in an expanded form for ease of access update unlike Unix type file systems. Unlike GFS, metadata is now stored in separate metadata stores.

Tectonic uses a blocksize that is of the order of KBs of data as per statistics [94].

8.9 Chapter Summary

The complete NFS v2.0 protocol specification is provided by Sandberg [104]. Several online and textbook resources describe the operation and implemenation details of NFS. See [111] for an example. NFS v3.0 and NFS v4.0 are part of RFC 1813 and RFC 7530, respectively. The Amazon EC2 file system is an implementation of NFS v4.0. There are several organizations that support various implementations of NFS; IBM, Oracle, NetApp.

Distributed file systems other than the ones that we described in this chapter include the Lustre file system [117], GPFS [105], ANY OTHER. Lustre and GPFS are POSIX-compliant whereas GFS is not. For this reason, both of these also maintain and manage metadata that may pose to be a burden for systems such as GFS and Tectonic that are workload-oriented.

In addition to designs of distributed file systems, there are several studies on how to store and search the metadata of highly scalable file systems. Some of the include [81, 87, 98].

A noticeable feature among the distributed file system designs that we summarized in this chapter is extendibility. Extendible systems allow for adding more resources such as disk drives in a seamless and transparent manner.

In recent years, there is an emerging set of applications that are leading to rethink on distributed file system and storage system design. In particular, traditional file systems supported an in-place write operation that allows the user to pick the offset at which to write the desired contents. New generation applications such as email stores, photo stores, rarely need to support in-place writes. Rather, such applications benefit from an append operation like the record-append that GFS introduced. Having record-append allows the file system to also order the updates to achieve consistency across multiple concurrent writes in addition to making it easier to version control. In addition, with these client facing applications, durability, availability, and consistency guarantees take higher priority.

Another noticeable difference in the way current generation users interact with file systems. Unlike the advanced users of the yesteryears who needed to interface with distributed file systems via operations such as mount. The current trend is to make the file system completely transparent to the user.

Coupled with the diminishing cost of storage per byte, some of the learnings with respect to system optimizations from the traditional era with respect to internal fragmentation and the like start to lose significance.

These changes in the conventional wisdom to new wisdom open up multiple opportunites in the design and implementation of distributed file and storage systems. Some of the insights into the design of highly scalable distributed file systems such as Colossus and Tectonic is at `https://queue.acm.org/detail.cfm?id=1594206` and also at `https://paulcavallaro.com/blog/facebook-tectonic-filesystem/`, respectively.

Questions

Question 8.1. Study the data structures that a typical Unix file system maintains. Assuming a block size of 2 KB, find the largest size a file can have in such a file system.

Question 8.2. Compare and contrast bewteen the various RAID levles

used in disk drives for data durability and the replication approach that distributed file systems use for achieving data durability.

Question 8.3. What is the potential trouble of deleting a file by its name in NFS. Will the troubles disappear if the protocol is stateful instead of being stateless?

Question 8.4. Consider the GFS Record-append operation or the write operation. Discuss what happens when one of the secondaries fails the write operation. Now, imagine that the primary waits for an acknowlegement from each secondary with respect to the write operation and asks all secondaries to retry the operation with a new offset. Describe how to complete this solution and discuss the guarantees this solution provides to the client for a recrod-append and a write operation. Compare this soltion with respect to the original solution in terms of the overheads/ consistency/ practicality/ and other considerations.

Question 8.5. Estimate the size of the metadata that the GFS master server has to hold in different scenarios: varying the chunksize and varying the number of files and their sizes. What are the limitations on the number of files and their sizes where the metadata is always held in the RAM?

Question 8.6. Discuss some advantages and disadvantages of the choice of having the GFS master server keeping the metadata in the RAM.

Question 8.7. Estimate the time taken to perform a write operation using GFS with a three way chunk replication, network statistics, and the data volume.

Question 8.8. Distinguish between the locking needed at the GFS master server and a Unix file system to create a new file.

Question 8.9. Discuss the differences between caching and replication including aspects such as usage, location of replicas, consistency issues, and the like.

References:

1. Managing NFS and NIS, Stern and Labiagga, Orielly, Second edition, 2001, Chapter 7.

Chapter 9

Distributed Databases and Transaction Processing

Databases have been around for several decades. The notion of relational databases gained immense popularity with the seminar paper of Codd [28] in 1970. Relational data model that Codd [28] proposed modeled a variety of scenarios and is also equipped with a efficient indexing, and a query language called the Structured Query Language (SQL) along with operations such as joins, that allowed an expressive way of storing and retrieving data. The notion of transactions and other measures ensured data integrity and enabled several applications. This model uses and assumes a single server and no replication of data.

This model sufficed so long as the data volumes were manageable by the ever-increasing server capabilities. With the growth of volume in data, this model struggled in various ways including scalability, performance, consistency, fault-tolerance, and the like. This led to the development of distributed databases and noSQL databases.

In this chapter, we study all these developments starting with the notion of SQL transactions. We will move to study commit protcols for distributed trnasactions, the development and designs of no-SQL databases, and some case studies in that regard.

9.1 Transactions

Recall that most computer architectures and operating systems do not gurantee that a process executes a sequence of instructions without being interrupted in between. While single user programs do not notice any difference

in this model of execution, programs that share variables are likely to experience difference in results based on differences in order of execution. In the context of databases, such a situation could occur when two different set of SQL statements get executed with arbitrary interleaving of instructions. Listing 9.1 shows a classical example of such a scenario.

Listing 9.1: Two Programs modifying shared variables

```
1  /* Program 1 */
2  .
3  .
4  balance1 = T.balance[1]
5  balance2 = T.balance[2]
6  balance1 = balance1 + 50
7  balance2 = balance2 -  50
8  T.balance[1] = balance1
9  T.balance[2] = balance2
10 .
11 .
12
13 /* Program 2 */
14 .
15 .
16 balance1 = T.balance[1]
17 balance2 = T.balance[2]
18 balance1 = balance1 - 100
19 balance2 = balance2 + 100
20 T.balance[1] = balance1
21 T.balance[2] = balance2
22 .
23 .
```

In Listing 9.1, we are updating two variables that correspond to the balance in accounts numbered 1 and 2. Assuming that the balance in accounts 1 and 2 before the start of Program1 and Program 2 is Rs. 1500 and Rs 2500 respectively, the final balance in these two accounts can differ based on when and how these instructions in Program 1 and Program 2 get interleaved in their execution and when these instructions take effect.

However, in the real-world programs of the kind shown in Listing 9.1 are not uncommon. In addition, unexpected errors can ensue while programs are in execution. For instance, the database server may crash while it is executing some parts of the two programs in Listing 9.1.

9.1.1 The Notion of Transaction

Guarding against such inconsistencies and errors requires gurantees on how the instructions across programs get exectued. This is where the notion of transactions comes in. Gray [55] introduced transactions in the context of database systems. SQL supports a mechanism to encapsualte a set of SQL statements as a transaction. These are via the `begin transaction` and `end transaction` keywords. Listing 9.5 shows the statements in Listing 9.1 as two transactions.

Listing 9.2: Statements enclosed as transactions

```
1  /* Program 1 */
2  .
3  .
4  begin transaction Transfer1
5  Update Table Accounts set balance = balance + 50 where
       acc-no = 1;
6  Update Table Accounts set balance = balance - 50 where
       acc-no = 2;
7  commit
8  .
9  .
10
11 /* Program 2 */
12 .
13 .
14 begin transaction Transfer2
15 Update Table Accounts set balance = balance - 100 where
       acc-no = 1;
16 Update Table Accounts set balance = balance + 100 where
       acc-no = 2;
17 commit
18 .
19 .
```

Encapsulating a sequence of SQL statements inside a `transaction` block ensures that those instructions get executed with certian gurantees. In particular, these blocks of SQL statements satisfy the conditions that: (1) either all of them are exectued or none of them get executed, (ii) the execution of these take the database from one consistent state to another consistent state, (iii) these instructions get executed in isolation as if no other instructions were executed in an interleaving fashion, and (iv) if these instructions are exectued successfully, all the effects of executing these instructions are

saved in permanent storage.

Harder and Reuter [58] gave the ACID mnemonic to the above four properties in that order. These stand for Atomicity, Consistency, Isolation, and Durability. Database systems guarantee that transactions follow the ACID semantics. To meet these four conditions, database systems incorporate a vaerity of mechanims. We will review these breifly in the following.

9.1.2 Atomicity

The atomicity property requires that either all the instructions that are part of the transaction get executed compltely or none of these instructions get executed. Each transaction has an outcome between commit or rollback. If the transaction has to be aborted for any unexpected reason, the rollback feature allows for rolling back the effect of the instructions executed so far. If no errors arise, then the transaction is moved to the commit state. A user or the system can move the transaction to the rollback state.

9.1.3 Consistency

For ensuring consistency, the database ensures that each instruction essentially starts with the database in a consistent state and moves the database to another consistent state. Note however that the database server executes instructions as programmers specify. So, by itself, the database server may not be able to enforce consistency in general. The database server can ensure that the state of the database is consistent before the instruction starts executing. It is the programmer's responsibility to ensure that the modifications to the database performed by the instruction ensure that the state is consistent after the instruction is executed. For instance, the programmer has to check for violations to business logic that do not take the database from a consistent state to an inconsistent state.

9.1.4 Islotaion

Isolation property ensures that the results of the updates of a transaction,s ay T_1, are visible to other transactions only on (successful) commit of the transaction T_1. In other words, the updates of T_1 happen as if these updates are happening in isolation to activities of the other transactions. To ensure that the isolation property is satisfied, one solution is to use locks. Databases support varying degrees and models of locking to cater to transactions.

One way to ensure isolation via locks is for the transaction to acquire a lock for every database object that the transaction wants to update. These

are called exclusive locks and transactions have to wait to acquire a lock if the database object they want to update is already locked by some other transaction. Naturally, this simple semantics can lead to deadlocks as the following example illustrates.

Consider just two transactions T_1 and T_2 and two database objects O_1 and O_2. Imagine that both T_1 and T_2 want to acquire locks to both O_1 and O_2. T_1 manages to get a lock on O_1 and T_2 manages to get a lock on O_2. Now, T_1 is waiting to get a lock on O_2 and T_2 is waiting to get a lock on O_1. This results in a deadlock. Database servers also, therefore, incorporate mechanisms to deal with deadlocks. These measures include deadlock prevention, deadlock detection, and deadlock recovery.

For deadlock prevention, one common strategy is to require that transactions acquire all the needed locks in an atomic manner. This however limits the degree of concurrency. Further, it is also not in general possible to identify what all database objects would a transaction need at the start of the transaction. Another technique is to impose an order on database objects that transactions have to follow in acquiring locks. This too can limit the degree of concurrency.

Deadlock detection is usually done by means of auxiliary data structures such as the wait-for graph that are also popular in the context of operating systems. Each transaction corresponds to a node in the wait-for graph. A directed edges from node u to node v in the graph represents that the transaction corresponding to node u is waiting for a lock that is currently held by the transaction corresponding to node v. A cycle in this graph indicates a circular dependency and hence a deadlock.

As transactions request locks, edges are added to the wait-for graph followed by a check for the presence of directed cycles.

Question: For how many locks would a transaction wait at any given point of time? Does this mean that the out-degree of the wait-for graph is at most 1?

Once a deadlock is detected, deadlock recovery mechanisms aim to ensure progress. One of the ways to do so is to break each of the cycles in the wait-for graph. This requires aborting a transaction that is part of a directed cycle in the wait-for graph. The choice of transaction to abort can be based on several factors such as the time spent by transactions, number of locks held/waiting, waiting time, and so on.

A second method to resolve deadlocks is to limit the amount of time a trasaction can hold a lock on a database object. In this case, a transaction holding a lock is aborted if it continues to hold the lock after a certain time elapses and some other transaction is waiting to acquire a lock on the same

object. This mechanism allows the waiting transaction to acquire the lock whereas the one whose lock is evicted/released has to be aborted. Notice that in this model, the eviction of a lock applies even if there is no potential deadlock.

Hiearchical locks is a way to increase the degree of concurrency in transaction processing. The hiearchical locking model that Gray [55] proposed envisions a hiearchy of database objects. One such support is via treating such as: the entire database, a table, a row, and the like. Yet another support is to also support a hierarchy in terms of the table structure based on the application requirement. For instance, in a database modeling the operations of a university, a table storing the overall Accounts information may be treated higher up in the hiearchy compared to Payroll information which deals with payment to employees. (The former may also deal with payments to vendors). In this setting, the database has to provide a semantic understanding of what various types of locks on each object mean for acquiring locks on other objects down the hierarchy.

9.1.5 Durability

Durability refers to the property that the effects of the transaction are recoverable even after failures. Here, failures refer to events that impact normal functioning of the system. These could be due to errors occuring at multiple levels: at the application/software level such within a running transaction that encounters a logic error, at the operating system or the system level such as failure of the volatile storage, deadlocks or resource unavailability, or at the stable media level including failures of disks. For the effects of successful transactions to persist, the database has to guard against the failures mentioned above.

Failures at the application or software level usually result in abort of transactions. However, the actions of such transactions up to the point of noticing an error and aborting are recorded in the log file so that these actions can be undone. Such undoing discards the effects of these unsuccessful transactions.

At the system level too, durability is acheived by using Write-Ahead-Logging (WAL) . In this model, the working log of the actions of the transactions are flushed to non-volatile storage *before* the commit of the transaction is acknowledged. During failure recovery, this log on stable storage helps in redoing the actions of the transactions that were committed. For failures of the stable media, the usual model employed is replication . In replication, data is usually copied into multiple independent storage devices

so as to prevent loss of data. The degree of replication is usually driven by application needs.

9.1.6 Nested Transactions

The above case of a transaction encapsulating a sequence of SQL instructions/statement is a simple setting. In general, it is possible to prepare nested transactions. A nested transaction consists of sub-transactions that are themselves encapsulated in another transaction. Nested transactions appear in many applications – consider for instance a user booking a holiday on popular travel websites such as MakeMyTrip[1]. The user chooses the travel mode and tickets, a place to stay, and books a set of other activities. Each of these are typically serviced by independent agencies and also view these as individual events instead of one combined set of activities. The user however sees all of them as a single unit. In effect, each of these individual bookings correspond to sub-transcations whereas the user transaction is the parent transaction. The website has to get a confirmation from each of these agencies before replying with a confirmation to the user. These semantics can be captured via a nested transaction. In general, it is possible that one of these sub-transactions may lead to another level of nesting due to the nature of entities involved. Therefore, nested transactions can have a deep parent-child structure with multiple levels of nesting.

In the case of a nested transaction, the rules for commiting or aborting apply to each sub-transaction. Each sub-transaction can be therefore committed or aborted on its own. However, the decision to commit is always a provisional decision since the parent can still abort all its sub-transactions, including those that wish to commit. A parent transaction has to wait for all its sub-transactions to reach a commit or an abort state. Only then, a parent transactions can choose to commit or abort. If a parent transaction aborts, all its sub-transactions, including those that make a provisional commit decision, will be aborted. If the top-level transaction commits, then all of the subtransactions that have provisionally committed can commit too, provided that none of their ancestors has aborted.

Notice that while it appears unsafe that a parent transaction being able to commit even though some of its sub-transactions may fail and rollback. However, there are setting where this behaviour is acceptable. Applying these rules to the earlier example, the user may not wish to purchase the trip if the travel tickets and hotel are not available, but may still go ahead if some

[1] www.makemytrip.com

of the activities are not available. This suggests that a parent transaction can choose to commit even though some sub-transaction aborts. So, the semantics of nested transactions allow the application to support a diverse set of applicaion requirements.

9.1.7 Commit/Abort Processing

9.2 Distributed Transactions

A summary of transaction processing in a centralized database as discussed in the earlier section opens up multiple challenges when we move to a decentralized, or distributed, database. In this setting, we envisage that a transaction accesses and updates data from multiple independent databases. A quick example is as follows. Consider using the now popular UPI where a user wishes to transfer an amount of Rs 100 from his account with the State Bank of India to the account of his friend with the Union Bank. The databases that these banks hold are spread across different administrative domains and are communicate via a network with its own fallibilties. Still, for the user, the UPI system has to provide certain guarantees on how these transfers can take place seamlessly and successfuly, at least in a large majority of instances. In this example, we have not even talked about how the UPI system itself that the user interacts with will recover from any errors or faults.

From the above example, it can be noticed that existing solutions that work for transactions in a single database do not carry over immediately to a transaction that work in a distributed setting. In particular, the semantics and mechanisms of how to arrive at commit or abort decision of transactions in the presence of failures requires a deeper study. In this section, we study some of these mechansims in this direction.

9.2.1 Model

The protocols that we study in this section make the following assumptions about the nature of faults that occur in distributed transaction processing. The system consists of $n \geq 1$ databases each of which is a likely *participant* in a distributed transaction. The participants are connected by means of point-to-point links (CLIQUE?). In addition to the database at each participant that is likey to encounter failures mentioned in Section **??**, there are additional failures that ensue. The links that connect the participants may fail. We however consider only crash or fail-stop failures in the sense

that once a link or a participant fails, it stops taking part in the execution of the protocol. Further, the failed entity is expected to be replaced soon after failure.

The algorithms or protocols do not tolerate arbitrary faulty behavior such as Byzantine faults where faulty participants have the potential to disrupt the outcome of the protocol by introducing spurious messages unlike the case of fail-stop failures. The system acts in an asynchronous mode of communication with messages getting delayed or lost in transit.

In the distributed setting, a transaction can access objects from databases that exist at more than one node and operate independent of each other. However, the goal for transaction processing is to still maintain the four properties of transaction namely, atomicity, consistency, isolation, and durability.

9.2.2 Commit Protocols

Before we discuss commit protocols in the distributed setting, let us briefly understand the failings of commit protocols from the non-distributed setting. Recall from Section **??** that the protocol from the non-distributed setting works as follows. Each transaction decides to commit or abort and informs the database (participant) accordingly. The participant acknowledges the decision of the transaction after taking into account any issues arising at the participant. WRITE ONE LINE ON WHEN THE DATABASE FORCES THE TRAN TO ABORT EVEN THOUGH THE TRAN WANTS TO COMMIT.

Henceforth, we will call this interaction between the transaction and the participant as a *phase* and the protocol from the above paragraph as a 1-phase commit protocol. We will now see how this 1-phase commit protocol falls short in the distributed setting. In the distirbuted setting, a transaction may involve multiple particpants and communication between participants is asynchronous.

A distributed transaction refers to a flat or nested transaction that accesses objects managed by multiple servers. When a distributed transaction comes to an end, the atomicity property of transactions requires that either all of the servers involved in the transaction commit the transaction or all of them abort the transaction.

In this setting, let us consider that one of the participants of a transaction acts as a *coordinator* . The coordinator has to make the final decision with respect to the outcome of the transaction. The coordinator can base this decision on the outcomes at other participants.

9.2.3 2-Phase Commit Protocol

Extending the 1-phase commit protocol, let us imagine that each participant in a transaction makes an independent decision to commit or abort the tranaction and sends this decision to the coordinator. The coordinator can then tally all these decisions and make a final decision. The coordinator relays this final decision to all participants and as per the 1-phase commit protocol, it marks the end of interaction between the participants and the transaction. Notice however that failures can render this solution inadequate. As the links may fail, the deecision that the coordinator relays may never reach one or more of the participants. In such a scenario, the coordinator does not know if all the participants acted accroding to the decision that the coordinator relayed. There are other such scenarios where this solution falls inadequate.

The failure scenario described above is part of a more generic problem that one often encounters in computer systems. This problem is referred to as the **2-Generals Problem** wherein two parties have to agree on a paritcular action by communication over a non-reliable channel. The channel may drop messages, or fail, and it is not possible to distinguish between the two scenarios. In fact, it is also known that there exists no solution to this problem if the two parties get only a

The above discussion indicates that indeed the 1-phase commit protocol is not useful in distributed transaction processing. A solution for distributed transaction processing is the *two-phase commit protocol* . This protocol allows the servers to communicate with one another to reach a joint decision as to whether to commit or abort.

Let us describe how the 2-phase commit protocol [54] works. A client starts a transaction by sending a request to a coordinator. This coordinator returns a transaction identifier (TID) to the client. Transaction identifiers for distributed transactions must be unique within a distributed system. The coordinator that opened the transaction becomes the coordinator for the distributed transaction and at the end is responsible for committing or aborting it. Each of the servers that manages an object accessed by a transaction is a participant in the transaction and provides an object we call the participant. The participants are responsible for cooperating with the coordinator in carrying out the commit protocol.

The protocol has two phases of communication between the participants and the coordinator. In the first phase of the protocol, each participant votes for the transaction to be committed or aborted. Once a participant has voted to commit a transaction, it is not allowed to abort it. Therefore,

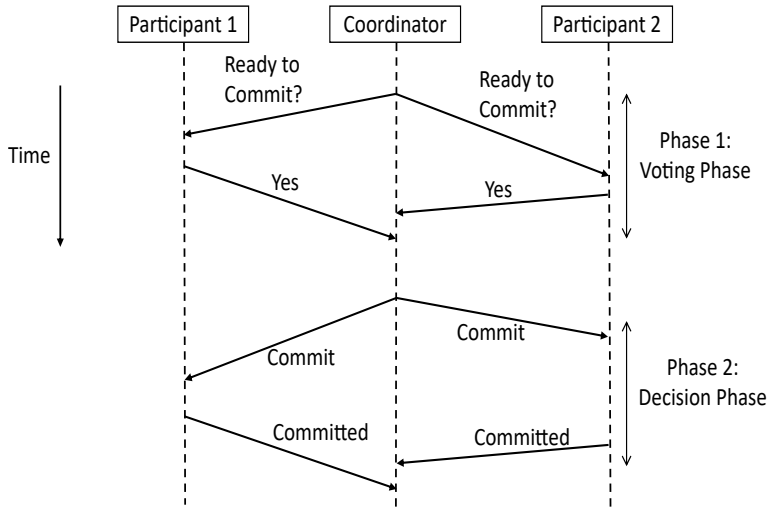


Figure 9.1: A typical interaction between the coordinator and two participants in a 2-Phase commit protocol.

before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to be in a *prepared* state for a transaction if it will eventually be able to commit it. To make sure of this, each participant saves in permanent storage all of the objects that it has altered in the transaction, together with its status as prepared.

In the first phase, the coordinator sends a message to each participant and asks if they are ready to commit. When a participant receives such a message from the coordinator, the participant replies with a Yes/No indicating its ability to commit to abort respectively. However, before ansewring Yes, the participant has to ensure that it records its actions in permanent storage so as to be able to recover from any errors that require the participant to restore its state correctly. If the participant answers with a NO, such a participant can abort immediately. The first phase is often called as the *Voting Phase* since this phase invovles the coordinator collecting the votes from the participants. See Figure 9.1 for an example.

In the second phase, the coordinator would tally all the votes received. If all the participant replies are recieved and all of them reply with an YES vote, then the coordinator can also decide to commit the transaction. If any of the participants votes NO and the coordinators receives a NO vite,

then the coordinator decides to abort the transaction. The coordinator communicates this decision to all the participants and the participants take the corresponding action with respect to the transaction. Participants that voted Yes are waiting for a doCommit or doAbort request from the coordinator. When a participant receives one of these messages it acts accordingly and, in the case of commit, makes a haveCommitted call as confirmation to the coordinator.

If there are no failures at the participants or the links, then the final decision are easy to arrive at and also implement. Failures at various stages of this protocol introduce complications in this protocol. In the following, we show that the protocol can withstand these failures and *eventually* halts even though the time it takes to halt depends on the failures that the protocol encounters in its execution.

Now consider various points at which failures can show up and see how the protocol addresses such failures.

Suppose that one or more of the participants does not get the message from the coordinator that asks participants to inform the coordinator of their decision to commit or abort the transaction. In this case, the coordinator will get fewer votes than needed. At this point, the coordinator does not know if the participant is just delayed in sending its reply or the message from the coordinator did not reach the participant. The protocol suggests that in this case, the coordinator abort the transaction and send an abort message to all the participants. Participants who voted Yes in response to the coordinator's message in Phase I also have to abort the transaction now.

In another likely failure, suppose that all the participants send their vote to the coordinator in Phase I. The cooridnator now fails to run the second phase. In this scenario, the participants who voted to commit the transaction wait for a message from the coordinator to either commit or abort the transaction. The participants have an option to send a message to the coordinator enquiring about the status of the transaction. At this stage, the state of the transaction at such a participant is *uncertain*. If the coordinator really did undergo a failure and recovers from the failure, the coordinator eventually sends a decision. Till then, the state of the transaction at the participant that voted to commit is uncertain.

Another situation that could occur is when a participant is ready to commit but has not heard from the coordinator in Phase I. In this case, the participant continues to wait for the message from the coordinator. If the application wants to close the transaction and waits for an outcome (commit or abort), the participant can choose to abort the transaction. If the message from teh cooridnator eventually reaches the participant, the

participant can send a reply indicating abort.

Notice from the discussion that the protocol tolerates a sequence of failures while the outcome of a transaction is yet to be decided. In the best case scenario where there are no failures and p participants in a transaction, the protocol requires $3p$ messages. In scenarios with failures, the number of messages and the time taken to arrive at a decision increase.

Extension to Nested Transactions

Recall that in a nested transaction, a transaction can have multiple subtransactions with a nested structure. The 2-phase commit protocol can be naturally extendted to nested transactions also. The details follow.

Let us call the outermost transaction in a nested transaction as the top-level transaction. All other transactions are called as sub-transactions. Upon completion, each subtransaction can make a decision on its own to either commit provisionally or to abort. A provisional commit indicates that a subtransaction has finished correctly and will probably agree to commit when it is subsequently asked to.

There are two variants of 2-phase commit protocol in the nested setting. In the hierarchical 2-phase commit protocol, the protocol runs as a multi-level protocol starting with the top-level transaction. The top-level transaction acts as the coordinator initiates the protocol by sending the *cancommit* message to its child subtransactions. These subtransactions for which the top-level transaction is the parent act as the participants in this interaction. These subtransactions relay such a message to their children subtransactions and act as a coordinator for that exchange. These messages are sent down the hierarchy of the nested transaction structure. Each parent acting as a coordinator collects the replies of its children subtransactions and relay these to their parent for an eventual decision which is again sent to the entire subtransactions.

In the flat 2-phase commit protcol for nested transactions, the top-level transaction acts as a coordinator and treats all other subtransactions as participants. The top-level transaction initiates the 2-phase commit protocol with all the subtransactions.

9.2.4 3-Phase Commit

Even as the 2-phase commit protocol eliminates some of the limitations of the 1-phase commit protocol, it still suffers from a few lacuane. In particular, the 2-phase commit protocol is a blocking protocol. Hence, once Phase

I is complete, participants have to wait for a message from the coordinator before they take any unilateral action to commit or abort the transaction. With cascading failures and network partitions and possible recovery of the coordinator delaying the process of arriving at the final outcome, such a blocking wait can hamper the participants ability to process other transactions. Locks on objects held by this transaction cannot be released until the participants receive the final outcome from the coordinator. Such a long wait can result in decreased throughput for transaction processing at participants.

To overcome this limiation, Skeen [109] proposed a quorum-based commit protocol called as the 3-phase commit protocol . Before we describe the 3-phase commit protocol, we first outline the assumptions that the protocol works with.

9.2.5 Assumptions

We assume that all the partcipants have point-to-point communication links and that this network may not deliver messages in the order they were sent and does not detect lost messages. The network will not introduce spurious or spontaneous messages, and that garbled messages can be detected. We say that the network suffers from a partition if there exist more than one disjoint groups of participants such that there are no communication links from participants in one group to those in another group. Each such disjoint group consititues a partition of the network.

Similar to the model studied in the case of 2-phase commit protocol, we continue to deal with distributed transactions. We imagine that the transaction T consists of a set of sub-transactions $T_1, T_2, \cdots, T_p$ where p is the number of participants in the transaction. Any subtransaction can be unilaterally aborted. For transaction T to commit, all participants of T must agree to commit their subtransaction.

9.2.6 The Protocol

The 3-phase commit protocol allows quora to proceed to decide the final outcome of the transaction. To this end, the protocol maintains two types of quoras: a commit-quora and an abort-quora. Each participant also gets a number of votes which is an integer that is at least 0. (The number of votes is 0 if the participant is a passive participant.)

We use letters V_T, V_C, and V_A to denote the total number of votes with each participant, the number of votes required for a commit quorum, and

the number of votes required for an abort quorum, respectively. For the commit protocol to be resilient, it must hold the following three conditions:

1. $V_C + V_A > V_T$ and $0 < V_C, V_A \leq V_T$.
2. When any participant is in the state of commit, then at least V_C other participants are in the state of commit.
3. When any participant is in the state of abort, then at least V_C other participants are in the state of abort.

The last two conditions insist that before any irreversible decision is made, then a guranteed minimum number of participants agree on the decision. This condition can be maintianed only if the following two statements hold:

2.a For the first participant to commit, a commit quorum of participants must be in the state of commit, and

2.b After any paritcipant commits, the commit quorum must be maintained.

The following lists a similar set of conditions for Condition 3.

2.a For the first participant to abort, an aboft quorum of participants must be in the state of abort, and

2.b After any paritcipant aborts, the abort quorum must be maintained.

Notice that the 2-phase commit protocol does not satisfy Condition 2. (See also Question AA). In fact, it can be argued that any protocol where there is a single commitable state for each participant, cannot satisfy Condition 2. The 3-phase commit protocol, being a quorum based protocol, therefore adds one additional state called the *prepared-to-commit* state.

Similar to the 2-phase commit protocol, for every transaction, there is a coordinator that directs the actions of the participants. The protocol has three phases described as follows. In the first phase, the coordinator asks each participant if it is ready to commit. Each participant sends a reply to the coordintor with an Yes nor No with Yes corresponding to readiness to commit and No corresponding to an abort. In the second phase, the coordinator checks if all participants from whom a reply is received at hte coordinator have replied with an Yes. If so, the coordinator informs all

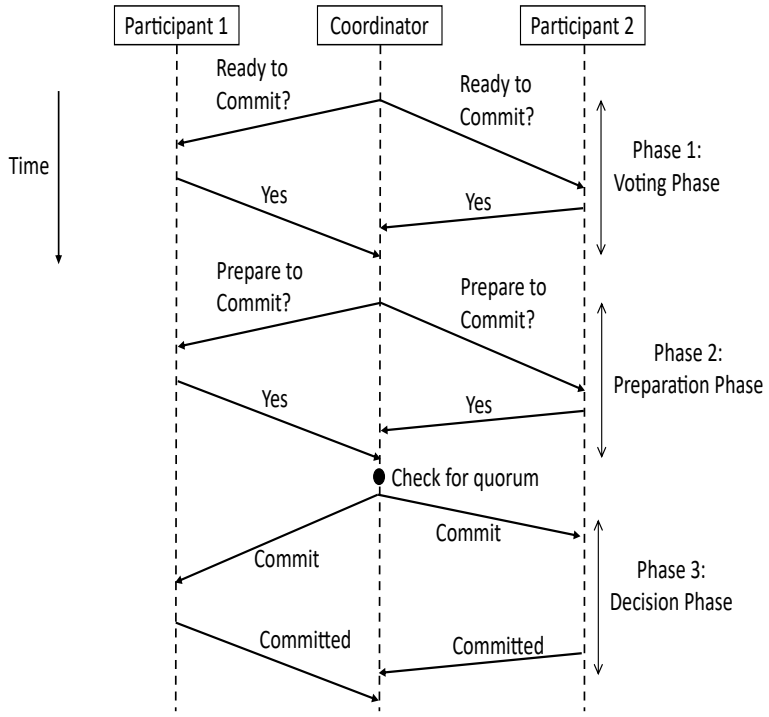


Figure 9.2: A typical interaction between the coordinator and two participants in a 3-Phase commit protocol.

participants to move the prepare-to-commit state. If any participant sends a No to the coordinator in the first phase, the coordinator sends an Abort message to all participants in this phase.

In the third phase, the coordinator checks if there is a commit quorum. If such a quorum of size at least V_C is available, then the coordinator confirms that participants can commit and sends a message accordingly. If such a quorum is not yet established, then the coordinator must not have heard from all participants in the prior phase. In this case, the coordinator waits to hear from other participants.

Notice from the above description that the protocol does not expclitly build an abort quorum. Since any one participant aborting will mean that all partcipants are asked to abort, it holds that no participant will enter the prepared-to-commit state in this scenario establishing the two conditions needed for abort quorums to be satisfied.

Recovery from Failures

The earlier description does not address how the 3-phase commit protocol recovers from failures including network partitions. There are two main steps in this recovery process. Consider that a network partition is induced due to faults. In this case, we imagine that multiple sets of participants idnetify the network partition. Each such set of participants can then run the same commit protocol on their partition by forming a quorum and arrive at a final decision for the transaction. Skeen [109] calls this protocol as the termination protocol.

Question: There will not exist two partitions where one has the commit quorum and the other has the abort quorum.

If such a quorum does not exist in any partition, then the participants wait until a quorum can be established. For such a quorum to be available, it must be the case that some partitions merge to form a bigger partition due to repair in the network partitions. Skeen [109] calls this protocol as the Merging protocol. Once the participants merge to form a bigger partition, they try establishing a quorum again. Finally, recovery of any participant or the coordinator from a failure can be treated as a case of merging an isolated partition.

Termination Protocol In the termination protocol, the main goal is to have a final outcome for the transaction. To this end, the first step of the termination protocol is to elect a (new) cooridnator for the partition. This step can be accomplished by making use of existing leader election

algorithms, **cf.** [**?**] for instance. Indeed, the 3-phase commit protocol works even when the leader election algrorithm chooses more than one participant as the coordinator. The second step of the termination protocol is to mimic the actions of the 3-phase commit protocol. There arise additional challenges compared to the description provided in **??** as the new coordinator does not have the same information as the original coordinator. Morever, depending on the outcome of the first step of the termination protocol, there may be multiple coordinators working in different partitions or in a common partition.

Nevertheless, the job of the coordinator is to check if there exists a commit quorum or an abort quorum. This involves the exact three phases of the commit protocol, but is now performed by the new coordinator and the participants in a partition. In the first phase, the coordinator asks the participants of their state of the transaction. If any of the participants already is in the state of commit (abort), then the coordinator informs all other participants to commit (abort). Such a commit at some participant could have happened before the failure ensued, but there must have been a valid commit quorum at that time. If not, the coordinator has to see if there is a commit (abort) quorum based on the replies received at the coordinator.

A commit quorum is a set of participants such that at least one of them is in the prepared-to-commit state and the sum of weights of the participants in the prepared-to-commit state plus the weights of the participants in the `wait` state is at least V_C. If such a commit quorum exists, the coordinator will inform all the participants in the `wait` state to move to the `prepared-to-commit` state and subsequently commits the transaction unless additional failures ensue. A similar procedure can be stated for the existence of an abort quorum and when the coordinator decides to abort the transaction.

Additional failures during the termination protocol can delay the decision of the coordinator. In this case, the protocol blocks if there does not exist a commit quorum or an abort quorum.

Case study/Question: Consider the commit protocol used in UPI. Notice also that the majority of transaction in UPI deal with at most 2 or 3 participants. So, is there a way to design a better commit protocol?

The Merging Protocol This protocol applies when a network partition is repaired and failed communication links are restored. Let us assume that some underlying protocol such as periodic pings detect the restoration of

failed links.

In the merging protocol too, a new coordinator has to be elected. There are multiple ways to accomplish this. One possibility is to let the new coordinator be one of the coordinators from the partitions that are now merging to form a bigger partition. The new coordinator now executes the 3-phase commit protocol as mentioned earlier.

Question: Can there be multiple quora of the same type in the 3-phase commit protocol? What type? Will this impact the consistency needed?

9.3 Relaxations to Distributed Transaction Processing

The earlier section described commit protocols for distributed transaction processing. With additional details, it is possible to ensure that distributed transactions also satisfy the ACID constraints typically imposed on transactions. However, this consistency comes at a price. As noted earlier, the absence of a final decision on the status of a transaction can hamper the throughput of participants. Going forward, we therefore want to see if the strict ACID semantics can be simplified in the distributed setting.

We will start with an example. Consider the setting where we make digital payments via the UPI interface. Some vendors also support a portable speaker device that plays an audio message after every successful transaction. These audio messages inform the recipient that the transfer of money is successful and eliminates the need for the recipient to check the status of the transaction. This facility is quite useful for recipients who deal with a large volume of transactions and in bursty situations.

In the above setting, consider the pseudocode of a UPI transaction. If the audio message is also part of the transaction, as Listing **??** shows, strict ACID semantics apply to that part of the work of the transaction also.

Listing 9.3: A sequence of SQL statements spanning mltiple tables

```
1 Begin Transaction
2     insert into UPI_Transaction(tran_id, from_id, to_id,
          amount)
3     Update Accounts_From set balance -= amount where
          user_id =
4                                from_id
5     Update  Accounts_To  set balance += amount where
          user_id = to_id
```

```
6       insert into Audio_Server(tran_id, from_id, to_id,
            amount)
7   End Transaction
```

Now, notice that the system dealing with the audio message has multiple failure points. In Line aa, the entry into the queue at the server is a concurrent queue since there will be multiple writers to the queue. So, there are chances of failure to add the required entry to the queue. Further, the queue could be full resulting in a failure to the add operation. The network link between the coordinator and the queue server may also fail.

From Listing **??**, we note however that the bulk of the transaction is complete by Line aa. So, it would appear that it would be practical to decouple the audio messaging part from the transaction. This may lead to inconsistencies since it is not clear as to what messages have to be pushed to the audio queue server. Moreover, if the purpose of the audio message is to provide real-time updates of successful transactions, it is not practical to make this a batch job that pushes all the messages from a stored database at the end of the day or so.

This situation calls for a solution that meets the practical concerns as well as goes beyond the strict ACID semantics. One possibility is to decouple the bulk of the transaction from the udpate to the audio service. If the main transaction is successful, then as Listing **??** shows, we can launch another transaction to update the audio server.

Listing 9.4: Separating the statements from Listing **??** with separate transactions

```
1   Begin Transaction
2       insert into UPI_Transaction(tran_id, from_id, to_id,
            amount)
3       Update Accounts_From set balance -= amount where
            user_id =
4                                   from_id
5       Update  Accounts_To  set balance += amount where
            user_id = to_id
6   End Transaction
7
8   Begin Transaction
9       insert into Audio_Server(tran_id, from_id, to_id,
            amount)
10  End Transaction
```

The solution in this case as presented in Listing **??** does have semantic differences with that of what Listing **??** does. For instance, between the two

transactions in Listing **??**, if the first transaction succeeds, the second may still fail. In such a case, Listing **??** does not ensure that the audio message will be ever reaching the server.

A better way to resolve this problem is to ensure that we use a persistent data structure such as a persistent queue to couple the events across the two transactions. The modified listing is in Listing **??**.

Listing 9.5: Continuing the example of Listing **??**

```
1  Begin Transaction
2      insert into UPI_Transaction(tran_id, from_id, to_id,
           amount)
3      Update Accounts_From set balance -= amount where
           user_id =
4                                from_id
5      Update  Accounts_To  set balance += amount where
           user_id = to_id
6      Queue message (tran_id, from_id, to_id, amount);
7  End Transaction
8
9  Begin Transaction
10 For each message in Queue do
11     Begin transaction
12         insert into Audio_Server(tran_id, from_id, to_id,
               amount)
13 End Transaction
14 End-for
```

From the listings presented in Listing **??**–**??**, we note that the it is possible to relax the ACID constratins at time for practical reasons. We get more efficiency in some situations by doing so, at the cost of reduced consistency. This state of reduced consistency can be made to be a state of *eventual consistency* by using additional mechanisms such as persistent data structures, idempotent updates, and the like. However, the database itself has increased availability even at the expense of its state being not fully consistent. This semantics of distributed transactions is also known by the acronym BASE which stands for Basically Avaiable, Soft State, and Eventually Consistent.

Such a relaxation of the ACID semantics brings other benefits also to distribtued databases and transaction processing. Some of these benefits include the increased availabilty of the database and strong support for scalability. In particular, databases now have greater flexibility in using features such as sharding and partitioning with multiple servers maintaining

parts of the database. This allows for separate servers to maintain and update independent pieces of data. In addition, database designers can use other strategies such as replication to increase data availability even as the state of the replicas may not always be identical at all times.

Incidentally, this relaxation coupled with scalability also paved way for the ability of systems to support large volumes of data – at the scale that "big data" deals with. The scale-out of databases to having multiple nodes coordinate to store the data leading to distributed databases systems is yet another development that resulted from this. Distribtued databases were able to offer a scale and performance that a traditional single server relational database system could not acheive. These databases also came to known as *no-sql* databases .

Many current generation distributed databases follow the BASE semantics. These include the Bigtable from Google, Dynamo from Amazon, and other applicatio specific databases such as MongoDB and Neo4J for documents and graph databases, respectively. In the following section, we will summarize the salient features of some of these databases.

9.4 Google's Bigtable

Bigtable from [24] is Google's database system which can be essentially thought of as a big, sparse, distributed, and persistent sorted map that can store multidimensional structured data. Bigtable is by design able to handle storage of petabytes of data and to thousands of machines. Bigtable does not adhere fully to the relational database model and hence moves away from all the constraints that come with relational database. Bigtable, instead, allows users to control the data layout and format. Data is primarily stored as arbitrary strings arranged and indexed as rows and columns. As we will see in detail, the simple design of Bigtable makes it flexible to be useful in a variety of applications including the traditional throughput-oriented batch processing jobs to modern user-centric latency-sensitive applications.

9.4.1 The Bigtable Data Model

In its simplest form, one can think of Bigtable as a distributed hash table. The key is a 3-tuple of row index, column index, and a timestatmp. Each of these tuples themselves is a string. The value is a string. This simple design comes with great flexibility depending on a variety of needs of applications.

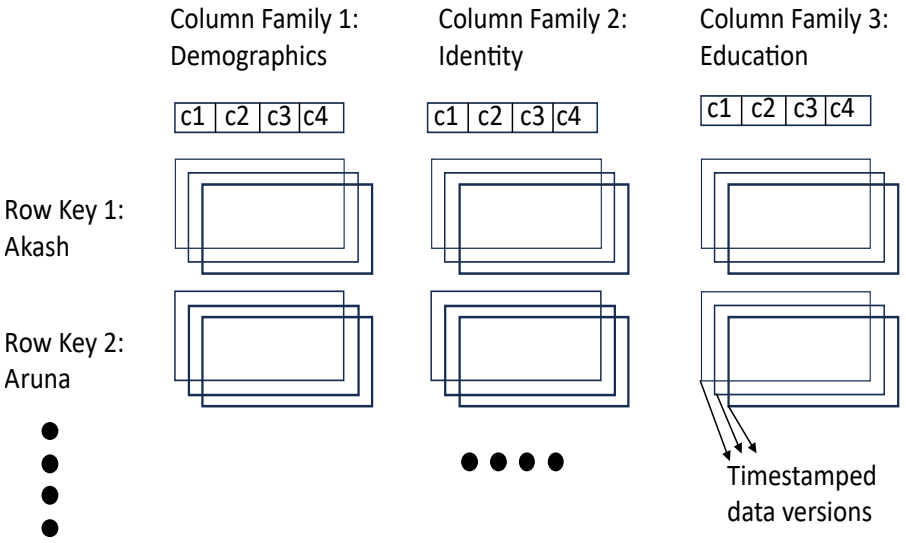


Figure 9.3: A typical Bigtable with three column families.

Rows

Each of the row indices in Bigtable is a string with a size limit of 64 KB. Irrespective of the number of columns in any row, Bigtable ensures that the read and write of a single row is done in an atomic manner.

In addition, Bigtable keeps data in lexicographic order by row keys. The entire set of row keys of a table are partitioned into ranges with each range called as a *tablet* . Tablet is the unit of storage, distribution, and load balancing across machines. It therefore holds that reading a contiguous set of row keys is usually serviced at a single machine and saves on communication cost.

Columns

Bigtable allows columns to be grouped into a *column family* . Each column is therefore identified by the column family it belongs to and the *column qualifier* within the column famliy. The column qualifier is unique within the column family. A column family has to explicitly created before creating columns under that family and storing data under such column names,.

Timestamps

Notice that each key in the table is the 3-tuple of row, column, and timestamp. The first two tuples, row and column values, denote a cell. Entries within a cell are distinguished by the timestamp tuple. Figure **??** shows an example. Even as there can be multiple column families defined for each row, each of which contains multiple columns, Bigtable allows for the possibility that some of these columns do not contain any information. This leads to the sparsity of the table and as we will see later, empty cells do not add to the storage space.

Bigtable has two ways to assign timestamps to cells. Timestamps are set to the real clock time in microseconds by Bigtable. This however may not guarantee uniqueness of timestamps. Applications that need fully unique timestamps can use their own mechanisms to set the timestamp to unique numbers. Within a tablet, cells with the same row and column key but with different timestamps are stored in reverse chronological order (latest timestamp first).

As the number of cells may grow adding to the data volume over time, Bigtable supports two simple mechanisms to truncate, or garbage collect, old cell versions. One such mechanism is to keep only a number of versions of the cell that the client can specify. Another is to enforce that cells corresponding to a prespecified duration be kept and the rest be discarded.

The timestamp tuple in the key allows for keeping track of the changes to the value at a given row and column over time. This feature is particularly useful for applications that track the change of data over time. For instance, if the data is the exchange rate between two currencies, then the latest exchange rate and also the past exchange rates can be read by means of a row and column index. Similar examples exist in emerging areas such as IoT systems, time-series data based applications, and the like.

9.4.2 Bigtable Architecture

The earlier section detailed how a table is to be viewed in Bigtable. Now, we move to discuss how Bigtable stores tables. The basic building block of any table in Bigtable is a *tablet* which is a contiguous set of row keys. Each table starts with one tablet and as more data is added to the table, more tablets are created by splitting an existing tablet into multiple tablets. Each tablet is usually kept to a size of few hundred MB. Google implementation set this value to be 100-200 MB.

Each tablet is stored internally in Bigtable using the Google File System

(GFS) as an SSTablet . An SSTable, (short for Sorted String Table), is a persistent and ordered immutable hash table. As mentioned earlier, the key is the 3-tuple of (row, column, timestamp) stored internally as a string, and the value is an arbitrary string. Each SSTable is stored as a sequence of blocks, each of which is of size 64 KB and is configurable. Every SSTable stores the list of blocks that belong to the SSTable as an index and this index is stored at the end of the SSTable. Bigtable loads this index into main memory when an SSTable is opened. By doing so, it is possible to perform a lookup on the index to locate the required block to be eventually read.

Bigtable stores SSTables in machines arranged as clusters. The clusters are expected to be managing their own mechanisms for scheduling and machine failures within the cluster. Each machine in a cluster is also called as a *Tablet server*. New tablet servers may be added to an existing cluster, and existing tablet servers may be removed from a cluster to adjust to the load and other conditions. Each Bigtable cluster stores a set of tables where each table consists of a set of SSTables and each SSTable contains all data associated with a row range.

The Bigtable architecture uses a single master node, also known as the *Master server*. As in GFS (**cf.** 8.5), most client operations do not flow thorugh the master node. This removes the throughput bottleneck at the master node. Clients communicate directly with tablet servers for tablet location, reads, and writes. However, the master server handles the assignment tablets to tablet servers, detecting the addition and removal of tablet servers, load balancing of tablets at tablet servers, creating column families, garbage collection of files, and handling changes to table schema.

The tablet server manages a set of tablets including the read and write requests to the tablets and also splits tablets that have grown too large.

Locating a Tablet

Recall that Bigtable clients do not communicate with the master even for locating the tablet server that stores a required tablet. We now outline the procedure clients follows to locate a tablet. Bigtable uses a *Root Tablet* that contains the location of all tablets. This location information is stored in a table called the *METADATA Table* . Since the metadata table is a table in Bigtable, this too is arranged as tablets. Each such tablet contains the location of a set of tablets corresponding to all user tables. The root tablet is the first tablet in the Metadata table and is never split.

The tablets of the Metadata Table are themselves stored as a B+tree

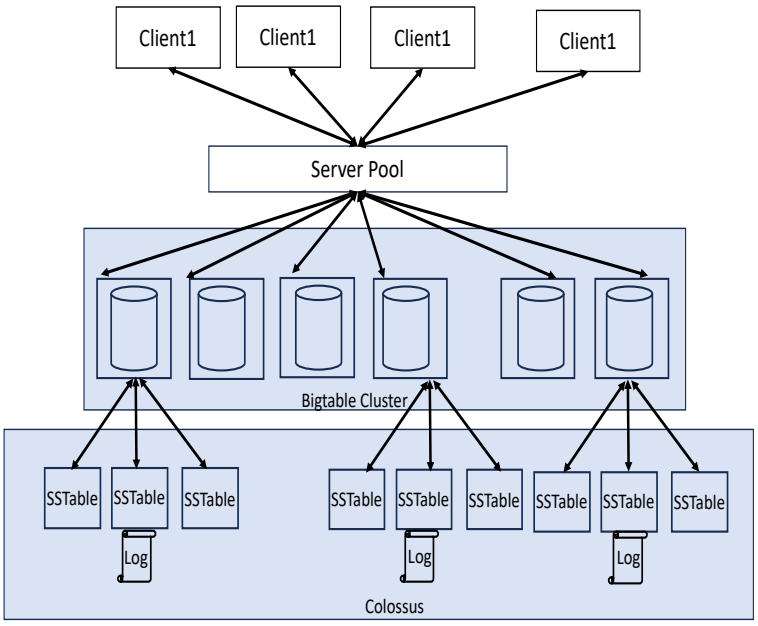


Figure 9.4: The architectural view of Bigtable.

structure [16] connsisting of three levels. This three level structure with reasonable size on the Metadata Tablet and row, can accommodate typical Bigtable system requirements. See also Question AAA.

In this three level hieaerchical arrangement of tablet location information, a client that is seeking the location of a tablet will need up to three round-trip accesses for a fresh request. Clients can cache the tablet location information and benefit from reduced round-trip accesses for cache hits. Cache misses, that can occur due to stale information in a client cache however may result in six round-trip accesses: the first three to realize that the location information is stale, and the next three to get the correct location information. Clients can also use prefetching of tablet location information to benefit from knowing the location of more than one tablet when reading the Metadata table.

Bigtable and Chubby

Bigtable relies on Google's distributed lock service Chubby [22] for a variety of purposes. For instance, the location of the Root tablet is a file stored in Chubby. Another use of Chubby is to ensure that there is only one active master server for Bigtable at any given time. Chubby is used to also keep

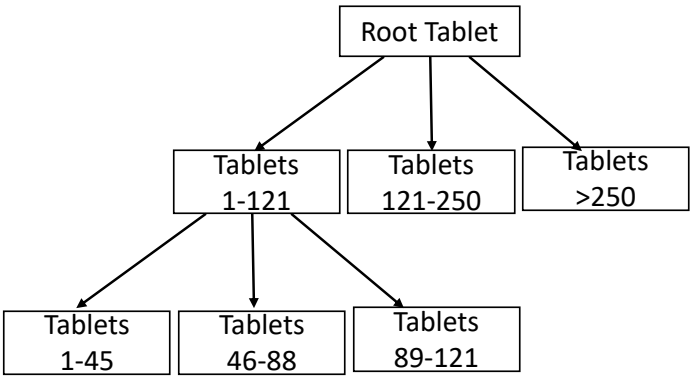


Figure 9.5: The architectural view of Bigtable.

track of the list of available Tablet servers. For all these tasks, Chubby helps by keeping a set of five replicas with one acting as the primary or the master. Other tasks that Bigtable uses Chubby for include storing access control lists, and storing schema information in the form of column families for each table, store bootstrapping location of Bigtable data, and so on.

The Chubby service is available so long as a majority of the replicas are running and can communicate with each other. If Chubby is unavailable for an extdended period of time, then, also Bigtable becomes unavailable. Internally, Chubby uses Paxos [80] to keep the replicas consistent. Chubby provides a simple namespace consisting of files and directories each of which can be locked by acquiring a lock.

Assigning Tablets to Tablet Servers

The master server keeps track of the set of available tablet servers and the current assignment of tablets to tablet servers. Each tablet is assigned to one tablet server. When the Master has a tablet that is yet unassigned to any Tablet server, the master makes assigns such a tablet to a tablet server that has space to store the tablet via a Tablet load request.

As mentioned in Section 9.4.2, Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named file in a specific Chubby directory. The master monitors this directory (the servers directory) to discover tablet servers. A tablet server stops serving its tablets if it loses its exclusive lock due to any reason, including a network partition. A tablet server will attempt to reacquire an exclusive lock on its file as long as the file still exists. If the file

no longer exists, then the tablet server will never be able to serve again, so it kills itself. Whenever a tablet server terminates, it attempts to release its lock so that the master will reassign its tablets more quickly.

The Master server has to keep track of the available Tablet servers and has to reassign tablets that were originally assigned to Tablet servers that no longer exists. To detect such unavaiable Tablet servers, the Master can continuously poll each tablet server for the status of the Chubby lock it holds. If the Tablet server reports that it does not have the Chubby lock any more, or if the master fails to poll the Tablet server, the Master assumes that the Tablet server is no longer avaialble. Tablets assigned to such a Tablet server are reassigned to other Tablet servers after the Master acquires a lock on the Servers Directory and deleting the entry corresponding to the unavailable Tablet server.

Operations at the Tablet Server

The tablet servers are responsible for performing write/udpate operations on the tablets. For a read operation, the typical steps involved are as follows. The operation is first checked for access rights and other properties. Once these checks are passed, the required data can be served from the correct SSTable. There is however one small issue that Bigtable has to contend with. Recent updates to an SSTable are often stored in an memory buffer called the *memtable* and these updates are pushed to the SSTables. So, the output of a read operation should include not only the relevant data from the SSTable but also the corresponding data from any applciable memtable entries. To this end, the Tablet server has to *merge* the relevant data from memtable and SSTable to create a merged view. The read operation can then be served from the merged view. Since the memtable entires and the SSTable entries are in lexicographically sorted order, the merge that also happens only once from memtable to SSTable is easy to perform.

A write/udpate operation has a few additional steps. As with a read operation, the first step is to check for access rights and well-formedness. Access authorization is checked by by reading the list of permitted writers from a Chubby file. The write is performed and is commited to a log file. Once the log file is written, the actual contents are inserted into the memtable. Data from memtable is over time merged with the corresponding SSTable. The presence of the log file helps Tablet servers recover a tablet if needed.

Compaction

Notice that the memtable is a buffer space in memory and will get full over time. So, Bigtable needs a mechanism to free up this space. To this end, when the current memtable reaches the size threshold, a new memtable is created and the full memtable is converted as an SSTable and written to GFS. This process is called a *minor compaction* and achieves two goals. Firstly, it shrinks the memory usage of the tablet server, and secondly, it reduces the amount of data that has to be read from the commit log during recovery if this Tablet server crashes. Notice from the discussion in Section **??** that read and write operations can continue even as a minor compaction is ongoing.

Every minor compaction therefore creates a new SSTaable but notice that during this compaction, merging of the data in memtable and data in SSTable is not done. If this situation is left unattended, then subsequent read operations have to merge an increasing number of SSTables to serve the relevant data thereby impacting performance. To address this situation, Bigtable employs another technique called a *major compaction*. This major compaction happens in the background and is triggered when there are a sufficient number of memtable entries to be merged with SSTables. During major compaction, deleted entries are also physically removed from SSTables allowing for an eventual reclaimation of space corresponding to deleted data.

Startup of a Master

Notice that Bigtable, like all distirbuted systems, is not immune to failures of various kinds including that of the Master server. In this case, a new Master server has to take the role of the failed one. The new Master has to also get the correct state information. This happens as follows.

The new Master acquires a lock in Chubby to establish itself as the only Master. This step prevents the proliferation of concurrent Master instantaions. Recall that Bigtable keeps files corresponding to active Tablet servers in Chubby. The new Master can therefore get the list of active Tablet servers. To proceed futher, the new Master corresponds with all the live Tablet servers to get a list of the tablets that are currently stored at the Tablet servers. In this step, the new Master has to check if it discovers the Root tablet. If not, then the Master treats the Root tablet as an *unassigned* tablet. The Master finally reads the Metadata table to know the list of tablets and uses this list to identify the list of unassigned tablets.

9.5 DynamoDB: Another Key-Value Store

Amazon Inc. [2] is another organization that deals with massive amounts of data across several applications. Amazon started with a distributed database system called SimpleDB [106] that prioritized availability over consistency. SimpleDB is a key-value store with *get()* and *put()* operations. SimpleDB however suffered from issues such as limited storage capacity for users, throughput of operations, and other performance issues. Moreover, for the kind of applications that Amazon deals with, it is quickly realized that the strong relational database model may come with multiple disadvantages with respect to scalability and availability. Traditional RDBMS models favour consistency over availability which limits the responsiveness of client-facing applications.

Amazon eventually improved on SimpleDB while retaining some of its features and developed a no-SQL style database system called Dynamo [108]. Alike GFS [51] an Bigtable [24], Dynamo is designed to work on commodity hardware and focusses on availability, scalability, and reliability, while also ensuring that a large majority of requests have a very small latency. In the rest of this section, we summarize the key properties and functionality of Dynamo.

9.5.1 Workload Aware Design and Assumptions

Dynamo (see also Section **??**) follows a workload aware design and is built on the following assumptions about the typical workloads. Most operations deal with a typical (key,value) mapping and do not span multiple data items obviating the need for strong semantics of relational schema and data model. This simplicity translates to a lot of design choices.

Since applications that use Dynamo are more interested in ensuring availability at the expense of consistency, Dynamo does not aim to satisfy the Consistency constriant in the ACID sytle transaction semantics. Moreover, as Dynamo applications use mostly single key updates, no guarantees are provided for isolation too. Dynamo ensures a notion of weak consistency.

Dynamo is used in controlled environements with a bigger degree of trust and security amongst nodes. However, since Dynamo is built out of commodity hardware, the design has to accommodate potential failures of components in addition to network failures that result in a partitioned network. These failures impact how the system reacts to dealing with uncertainty

[2] www.amazon.com

and availabiilty of data. Dynamo aims to increase availability by using optimistic replication techniques where changes to data percolate to replicas over time even as additional writes are always accepted in a partitioned and disconnected network. To this end, Dynamo adopts an eventually consistent state where all updates reach all replicas eventually.

Given that Dynamo allows writes that are not always consistent, there is always the question of how to resolve the conflicts and who resolves the conflits. Dynamo, owing to the workload aware design model, empowers applications (workloads) to resolve conflicts based on the application semantics. Applications can also choose to push conflict reoslution to Dynamo which uses generic and simple policies such as the latest write takes efffect.

Finally, based on the application requirements, Dynamo is designed to be scalable, dynamic, decentralized, and heterogeneity-aware. Scalable and dynamic designs support the addition or removal of storage nodes with little impact on the system performance and availability. Decentralized design allows for a system that can tolerate single point of failures and ensures that all nodes have a similar responsibility. The last of these requires Dynamo to be able to work with varying node capacities.

9.5.2 The Dynamo Architecture

As a distributed database, Dynamo offers several features and addresses several challenges including: partitioning, load balancing, scalability, failure handling and recovery, replication, concurrency, routing, version management, and so on. For many of these features, Dynamo relies on existing solutions developed in the distributed systems as mentioned below. For partitioning, it uses the approach of consistent hashsing **??** that we saw also provides good scalability. For version management and availability, it uses vector clocks, Section 2.4.2, with some changes. Dynamo also relies on gossiping and quorum based protocols for handling failures and providing availability under failures. We will detail these in the following.

The Dynamo Interface

Dynamo offers a very simple and minimal interface to applications with just two methods: the `get()` and `put()`. The `get()` function takes a key as an input and returns an object or a list of objects with conflicting versions along with a context. The context of an object contains the metadata about the object including information on the version of the object. The `put()` function takes as inputs the key, value, and the context and updates the key

with the provided value, updates the replicas, subject to the validity of the context provdied as part of the input.

Internally, the key and value fields are stored as byte arrays. The MD5 [102] hash function is applied on keys to generate a 128-bit identifier which determines the storage nodes responsible for the key.

Data Placement and Partitioning

Recall the consistent hashing scheme from Section **??**. Consistent hashing provides a simple yet effective mechanism to distribute objects across a dynamic set of hosts (nodes). Recall that according to consistent hashing, each node is mapped using a hash function f_n to a real number in $[0, 1)$ viewed as a ring, and each object is also mapped using a (different) hash function f_o to a real number in $[0, 1)$. The object is stored at the node that appears closest to the hash value of the object in the clockwise direction on the ring. Each node therefore stores all the objects that have a hash value between itself and the predecessor of the node on the ring. Nodes can be added dynamically in this model by effectively changing the set of objects that the nodes store.

Dynamo makes a few modifications to the consistent hashing scheme to address its requirements as used by Karger et al. [68] and Stoica et al. [112]. Each node is associated with mulitple positions on the $[0, 1)$ ring by using multiple hash functions. Each such position is treated as a virtual node. Each actual node is now a collection of one or more virtual nodes. This brings two advantages: one is to provide for better load balancing (see also Section **??**), secondly, virtual nodes provide support for heterogeneity.

Object Replication

One way to increase availability and ensure fault tolerance in a distributed database is to support replication. In this model, each object is stored as multiple locations (nodes). For a configurable parameter N, Dynamo stores each object as N replicas. Each key k is stored at N nodes with one of them being the *coordinator*. The coordinator is the node that is closest to the hash of the key k in the clockwise direction. The other $N - 1$ replicas are stored at the $N - 1$ clockwise successor nodes to the coordinator node. If some of these nodes happen to be the virtual nodes that correspond to the coordinator, these are skipped to ensure that there are $N - 1$ distinct physcial nodes that act as replicas. The list of distinct physical nodes that store a given object is called as the *preference list* of the object. Figure 9.6

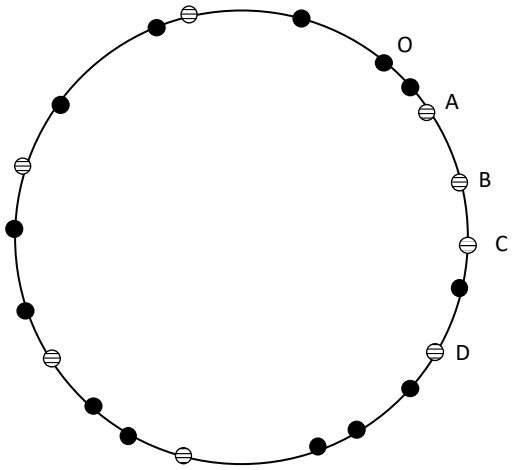


Figure 9.6: Dynamo

shows an example.

Version Management

Recall from Section **??** that a *put()* call can return to the caller even before the udpate has been propagated to all the replicas. In such a scenairo, a subsequent *get()* call may return a stale value, and node or network failures can mean that some of the updates may never reach all the required replicas for a long duration.

Dynamo keeps multiple versions of each object in the system at the same time. In a normal course of operation, older versions are subsumed (or overwritten) by newer versions. However, under certain circumstances such as concurrent updates and updates under failures, there could be multiple versions of the object that branch out from an older version.

Dynamo uses vector clocks, from Section 2.4.2, to track the different versions of an object. In Dynamo, the vector time of an object is the tuple of (node, counter). Each version of an object is timestamped with a vector clock. Given a object and two versions of the object, properties of the vector clock allow one to see if the two versions are causally related or not. Versions that are not causally related need to be reconciled either by the application or by Dynamo. In the former, notice that a *get()* on the object returns all the versions that are not causally related so that the application can perform reconciliation. In the latter,

Recall from Section 2.4.2 that one of the disadvantages of vector clocks is

the huge overhead of the vector time information that has to be piggybacked and maintained. To mitigate this overhead, Dynamo employ a truncation scheme that limits the disticnt number of vector time tuples on an object to a threshold value. Once this threshold is reached, the oldest timestamp is removed form the tuples. This truncation serves a practial purpose and in a worst case scenario lead to incompleteness and inefficiencies in reconciliation.

get() and *put()* in Detail

Given the additional details specified in the earlier sections, we now provide the mechanisms that Dynamo uses for the *get()* and *put()* operations. Assume for a moment that there are no intervening failures while these operations are under execution.

The node that handles a *get()* or a *put()* operation is always the coordinator node. Even if the request arrives at a node in the preference list of the requested object, this request will be further forwarded to the coordinator node. Dynamo uses the quorum mechanism (see also Section **??**) to ensure consistency and availability. In a quorum system, there are three configurable parameters, N, R, and W that correspond to the number of replicas, the size of the read quorum, and the size of the write quorum, respectively. These parameters are to be chosen such that $R + W > N$.

To execute a *get()* request, the coordinator requests all existing versions of the object from the N nodes in the preference list for that object. The coordinator waits to receive at least R replies. Once R replies are received, the coordinator returns all versions of the corresponding object that are not causally related. If there are indeed multiple divergent versions, the coordinator reconciles these divergent versions and writes a reconciled version that also superseded the latest version, thereby unifying the divergent versions.

The details for the *put()* request are similar with a few more steps. In particular, each put request has to be tagged with a vector timestamp. The coordinator generates the vector time for the put event and writes the new value to the local store. The information about the new value along with the corresponding timestamp is sent to the $N - 1$ nodes in the preference list. Each of these nodes send an acknowledgement to the coordinator. If the coordinator receives acknowledgements from at least $W - 1$ other nodes, the write is deemed successful.

Failures of nodes during the execution of the *get()* and *put()* operation are handled as follows. It is possible that the cooridnator has failed. So, the operation is handled by the node at the *top* of the nodes in the preference list of the object. In the *put()* operation, the update is sent to the top $N-1$

highest ranked reachable nodes in the preference list.

Just as with any quorum based system, a low value of R and W assures very high availability but at the risk of increased inconsistency. On the other hand, a high value for R and W may render the service unavailable at times. Most applications using Dynamo set N to 3, and R and W to 2. For applications that require further reduced latency for *put()* operations, Dynamo supports an optimization that works as follows. Nodes store details of the *put()* operation in a buffer at that are periodically written to Dynamo by a writer thread. Wtih this optimization, *get()* operations check the buffer to see if the object is in the buffer. If so, the object is read from the buffer and returned. If not, the object is read as in a normal *get()* operation.

Question: Can applications set W and R to 1 even with N say at 3? What does it mean for the application in terms of consistency?

Fault Tolerance and Handling Failures

Notice that Dynamo is desgined to be built out of comodity hardware. Such machines, and possibly the network links connecting these machines, are prone to failures. However, Dynamo aims to ensure a high degree of availability. From the execution details of the *get()* and *put()* operations, we note that both these operations require a quorum to be available. But, network partitions and other nodes failures can perclude the possibility of establishing a quorum. Dynamo employs the following techniques to address this problem.

Using Dynamo, applications can set the size of the quorum to as small as one. In this setting, the operations always succeed as long as there is at least one node that has the required object and the operation fails only if all the nodes in the preference list fail at hte same time.

Another technique that Dynamo uses is *hinted handoff*. In this case, to establish the required quorum, one of the nodes is handed a replica of the object on a temporary basis. This working node can be part of the quorum and stores such objects in a temporary local database. Once the node that failed comes back up, objects in this local database are sent to the original node and are removed from the local database. This process is called as a hinted handoff. The node chosen to keep the temporary replica is usually the one that is clockwise closest to the last node in the preference list.

The above techniques help address temporary or transient failures. Nodes in Dynamo may experience more permanent failures requiring the replacement of the failed node. In this scenario, soft mechanims such as hinted handoff will not work since the failed node may be unavailable for a sig-

nificant duration. These permanent failures are handled with the following mechanisms.

Each node maintains a Merkle tree [89] of the range of keys that it handles. nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common and determine if they have any differences and perform the appropriate synchronization action.

Frequent but transient failures of nodes can result in data reabalancing that adds overhead to the system. To reduce the number of data rebalancing operations triggered due to transient failures, Dynamo uses explicit addition or deletion of nodes. In this model, an administrator has to initiate a node add/remove command and these changes are commited to a persistent store. On change in the set of nodes, a gossip based protocol propagates membership changes to ensure an eventually consistent view of the nodes that are part of the system. Nodes in addition use periodic ping messages to know the state of each other.

9.6 Spanner and Distributed SQL

One of the limitations of Bigtable despite its scalability is the lack of support for transactions across multiple rows. This is also true of Dynamo. Both Bigtable and Dynamo prefer availability of the system over consistency and choose to sacrifice absolute (strict) consistency. While it is true that preserving strict consistency and satisfying ACID semantics on transactions does hurt application perfromance, it is debatable if that should really mean that the system will not even provide support for general purpose transactions. This lack of support for transactions coupled with other such issues meant that true distributed database systems that support general purpose transactions would be a worthwhile attempt.

In addition to the lack of support for distributed transactions, some of the other drawbacks noticed by application developers when using Bigtable include the key-value kind of data storage and lack of flexibility in supporting complex and evolving data schema. Google created *spanner* [29] which addresses that gap. In the following, we review the important features of spanner.

Basics of Spanner

Spanner is designed as a temporal multi-version distributed database where data is stored in schematized semi-relational tables. Data is assigned a version number upon each commit with the timestamp of the commit operation,

and older versions are garbage collected by suitable policies. Spanner, being distributed in nature, replicates data at multiple locations. Applications have the ability to configure the number of locations to keep replicas, the distace between the locations, and the locations where to keep the replicas.

Spanner uses Truetime to assign a globally consistent timestamps to commiting transactions. These timestamps satisfy the serialization order [60] that satisfies the following. If a transaction T_1 commits *before* another transaction T_2, then the commit timestamp of T_1 is smaller than the timestamp of T_2. Because of guaranteeing consistent timestamps, spanner can provide consistent reads and writes, consistent backups, consistent MapReduce executions, and so on. More details of this are in Section **??**.

Spanner exposes an SQL based query language and is designed to scale to millions of machines spread across hundreds of datacenters storing trillions of rows of data. Spanner also decides on certain aspects such as moving data across machines and possibly datacenters in response to load balancing and fault tolerance issues. These data movements are transparent to the application. Spanner stores shards of data across several machines spread over multiple datacenters. Any necessary resharding of data based on the addition or deletion of machines is also handled automatically.

Spanner uses other distributed system and database system building blocks such as Paxos [80] for achieving consistency, Colossus [61] to store data, 2-Phase commit **??** for distributed transaction processing, and 2-Phase locking [37].

Since its introduction in 2012, Spanner [29] has also undergone a few enhancements. While the original spanner retained some of the key,value style storage model, the work of Bacon et al. [11] moved the data model of spanner to a more relational database system with full SQL support. This support requires enhancing or reworking the query execution model, optimizations to query execution, storage model, and other backend tasks.

Spanner Architecture

The deisgn and organization of Spanner is centered around a *Universe Master* that in turn manages multiple *zones*, each with its own *zone master*, multiple *spanservers* and *location proxies*. Figure **??** shows these entities. Within a zone, the zone master assigns data to spanservers and the spanservers serve assigned data. Clients use location proxies to locate the spanserver that holds the required data.

The *Placement Driver* at the Unvierse Master interacts with spanservers and makes decisions with respect to data placement and migration. The

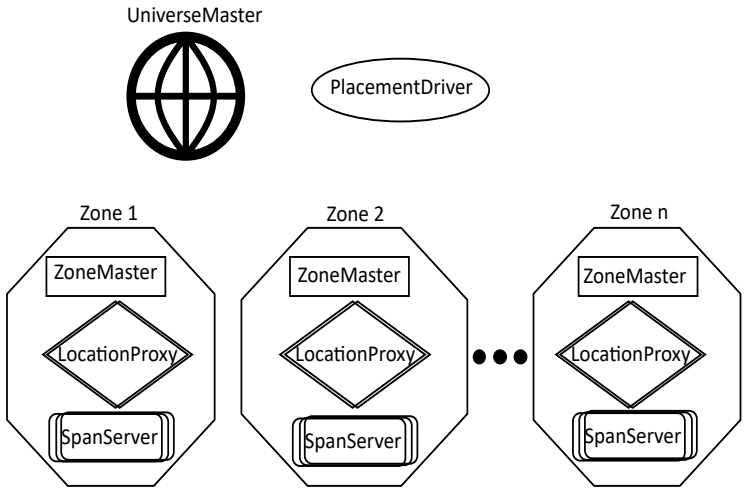


Figure 9.7: Spanner architectural diagram

Universe Master monitors the status of all the zone masters. Zones can be added or removed from the spanner system based on application needs. Zones acts as the unit of administrative deployment as well as physical isolation. Data can be replicated across zones based on administrative and application needs. A datacenter may have mulitple zones if the data from multiple applications needs to be on different zones (servers) within a datacenter.

We now move to describe the functionality of a spanserver.

The Spanserver Borrowing on the Bigtable [24], see also Section 9.4, abstraction, Spanner uses the notion of a *tablet* that stores mappings of the form (key, timestamp) $\rightarrow$ value. A set of such mappings is called a bag. Both the key and the valule are of type `string` and timestamp is a 64-bit integer. The timestamps serves to provide version to the data stored in spanner. Each such tablet along with its write-ahead log is stored in Colossus [61], see also Section **??**.

Each spanserver stores a set of tablets, roughly of the order of 100 to 1000 tablets. Replication is one of the essential functionalities of the spanserver. To this end, each spanserver implements a Paxos protocol on a per-tablet basis. Notice that in this model, every write is logged at two places: once due to the Paxos protocol, and once in the tablet log.

The Spanserver uses the Paxos protocol to support a consistently replicated set of mappings. The set of replicas of a bag are called its Paxos group

and each such group has a *leader* following the Paxos protocol. Writes to the bag start by initiating the Paxos protocol at the Paxos group leader. Reads access the data from the tablet at any replica that is (sufficiently) up to date.

At every replica that is a leader, the spanserver maintains two additional functionalies. One is *lock table* that records the state of locks and their ownership. Spanner uses two-phase locking [37] and each lock is for a range of keys. Operations that need a lock, for example transactions, check the lock table and wait to acquire the needed lock. Other operations such as simple reads do not use the lock table and bypass it. The second funcationality at a leader replica is that of a *transaction manager* that supports distribtued transactions using 2-phase commit [54], see also Section **??**. Transactions that involve only a single Paxos group can bypass the transaction manager. Transactions that involve multiple Paxos groups, one of the group leaders acts as the coordinator, also called as the *coordinator leader*, with the other acts as participants, also called as *coordinator slaves*. These group leaders execute a 2-phase commit protocol to arrive at the outcome of the transaction. The state of the transaction manager is stored in the corresponding Paxos group.

Spanner supports a semi-relational data model that has a mix of Bigtable data model and the relational data model. Applications create databases in a universe. Each database has a set of schematized tables. Each table has a set of rows with each row having mulitple columns with versioned values. Every table has a set of columns as primary key columns. This set of columns acting as the primary key also create a *name* for each row. Each table in spanner defined a mapping from the primary key columns to the non-primary key columns. An example of spanner schema is shown in Figure **??**.

As can be seen from Figure **??**, spanner database has a hierarchy of tables. This hierarchy is defined by the application in its scheme definition with the keyword INTERLEAVE IN. The table at the top of the hierarchy is called as the *directory table*. A row in the directory table with a key K along with all the rows in the descendant tables with key value K together form a *directory*. This interleaving of tables to form directories is an important feature of spanner. This allows applications to describe the locality relationships that exist across tables and helps improve performance in a sharded, distributed database. The phrase `ON DELETE CASCADE` in the schema defintion indicates that if the row with a given key K in the directory table is deleted, then all the rows in all the descendant tables with row value K are also deleted.

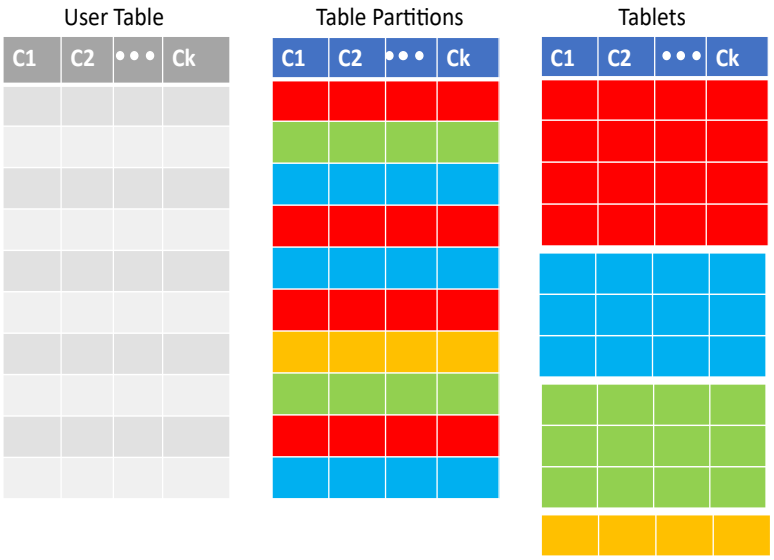


Figure 9.8: Spanner architectural diagram

Spanner treats the directory, as described above, as a basic unit of data placement and replication. Recall that multiple directories form a tablet. The notion of a tablet however differs slightly from that of a tablet in Bigtable. In spanner, the entries in a tablet need not correspond to a contiguous

Spanner and Truetime

One of the advantages that spanner has is to utilize Googles's Truetime [**?**] and its API to provide timestamps to operations within spanner. Recall from Section **??** that Truetime represents time as an interval where the interval shows the degree of uncertainty. Truetime API has three functions: $tt.now()$, $tt.before()$, and $tt.after()$. $TT.now()$ is the current time and retuns the interval $TTInterval.[earliest : latest]$. The other two functions can be suupored the TT.now() and are merely wrappers on $TT.now()$.

In particular, Truetime brings the advantage that any reads from the database at time t will see the effects of all transactions that have been committed by time t. In order to hold this gurantee, spanner has to assign a timestamp to each transation that commits by maintaining invariants that mandate how Paxos writes are timestamped and how the commit protcols works. These details are mentioned in the following.

Concurrency Control in Spanner

Spanner relies on the Truetime system and its API to support a variety of transaction processing models such as read-write transaction, snapshot transactions, and snapshot reads. Recall that snapshot transactions execute all their actions in the past but are still challenging to ensure consistency. In the following, we show how Spanner uses Truetime to assign timestamps to transactions and ensures ACID properties – equivalent to external consistency.

Read-Write Transactions Read-Write transactions include a mix of read and write to objects possibly across tables. We will consider the situation when indeed the transaction accesses objects across tables. Recall from the Spanner architecture that tables are stored as replicated directories. Below is the list of actions that Spanner does to process a Read-Write transaction.

- All read operations are directed to the leader of the corresponding replica group. The leader can acquire the read lock and subsequently serve the read requests. Each read request is served along with the timestamp of the data read.
- All write operations are buffered at the client before the transactions are committed. Clients use keepalive messages to ensure that the corresponding transaction is not closed due to reasons such as a timeout.
- Upon completion of all the reads and the writes are buffered, the client starts a two-phase commit. To this end, the client chooses one leader as a coordinator leader and sends a prepare request to all participant leaders.
- Every participant leader acquires the necessary write locks, chooses a prepare timestamp that is larger than the timestamps of any previous transactions, and logs a prepare record in its replica group through Paxos. Paricipant leaders also send a response to the coordinator leader with the prepared timestamp.
- The coordinator leader will send and replicate the commit record. Upon success, the coordinator will send the commit and its timestamp to all participants' leaders. Subsequently, a reply is sent to the client.

In the above, what is not clear is how a participant leader identifies a correct timestamp to be assigned. For this purpose, Spanner relies on Truetime and a few invariants that we explain below.

Let us define the lease interval of a leader to start at the time when it discovers it has a quorum of lease votes, and to end when it no longer has a quorum of lease votes. Spanner maintains the following disjointness invariant within each Paxos group which states that for each Paxos group, the lease interval of any Paxos leader is disjoint from the interval of every other leader. A Paxos leader can also release itself from being a leader. However, before giving up the leadership role, the current leader has to ensure that the maximum timestamp that the leader used is already in the past according to Truetime. In symbols, let $s_{\max}$ denote the maximum timestamp used by a leader. Such a leader can relinquish leadership only after $TT.after(s_{\max})$ is true. This ensures that a Paxos leader assigns timestamps that are strcitly within the lease interval of the leader. Further, when a leader assings a timestamp s, then it also advances $s_{\max}$ to s. By doing so, dsijointness is preserved.

We now briefly describe how the disjointness invariant is maintained. Essentially, in every Paxos group, the lease time of a leader and the next leader should be disjoint. This can be achieved if the current leader does a synchronous Paxos write indicating its lease time when it acquires the lease and subsequently whenever it extends its lease. A new potential leader can read these interval boundaries and know when it can acquire the lease. While this solves the problem, there will be these extra Paxos writes.

Using Truetime can help remove these additional writes as follows. Let us consider the situation that a leader ℓ, the ith leader of a Paxos group obtains its lease by getting the corresponding votes from the replicas in the group. Let $e_{i,r}^{send}$ denote the time that the ith leader sends its request to obtain leadership lease to replica r, the time this request is received at the replica r as $e_{i,r}^{receive}$, and the time at which replica r grants the request to the ith leader be $e_{i,r}^{grant}$. The leadership lease at the ith leader lasts up to $t_{i,r}^{end} := TT.now.latest() +$ lease duration where $TT.now.latest()$ is evaluated (called) after $e_{i,r}^{receive}$. The quantity `lease duration` is configurable based on the Spanner installation.

A replica r in the spanner system does not grant a vote to another potential leader in the same group until $TT.after(t_{i,r}^{e}nd)$ evaluates to true. Notice that the replica r may have failed in between and recovered, so it is important to log this fact that replica r granted a lease up to $t_{i,r}^{e}nd$. This log is written *before* the lease is granted by replica r via a Paxos write.

Let $v_{i,r}^{leader} := TT.now.latest()$ computed before e_i^{send} denote the time the potential ith leader can keep as a lower bound on the start of a lease vote from replica r. Notice that the potential ith leader must recieve enough

votes to become the ith leader. Let us denote the time at which the potential leader receives a quorum of votes as e_i^{quorum}. The actual leadership lease interval for the ith leader is the duration $lease_i := [TT.now.latest(), \min_r\{v_{i,r}^{leader}\}+$ lease duration]. The lease expires when $TT.after(\min_r\{v_{i,r}^{leader}\}+$lease duration) evaluates to true.

Finally, to show that the disjointness invariant is maintained at all times, we rely on the fact that the ith leader and a potential $(i+1)$th leader, $(i \geq 1)$, will have at least one replica $r*$ in common to the quorum votes received by them. This, coupled along with the fact that $r*$ would not grant a lease vote to the potential $(i+1)$th leader until $TT.after(t_{i,r}^{e}nd)$ evaluates to true, leads us to the following chain of inequalities that imply the disjointness invariant.

From the above, we note that the lease of the ith leader ends at time $lease_i.end := \min_r\{v_{i,r}^{leader}\} +$ lease duration. Now,

$$
\begin{array}{lcll}
\min_r\{v_{i,r}^{leader}\} + \text{lease duration} & \leq & v_{i,r*}^{leader} + \text{lease duration} & \\
& \leq & t_abs(e_{i,r*}^{send}) + \text{lease duration} & \text{by definition} \\
& \leq & t_abs(e_{i,r*}^{receive}) + \text{lease duration} & \text{by causality} \\
& \leq & t_{i,r^*}^{end} & \\
& \leq & t_abs(e_{i+1,r*}^{grant}) & \\
& \leq & t_abs(e_{i+1,r*}^{quorum}) & \text{by causality} \\
& \leq & lease_{i+1}.start &
\end{array}
$$

Coupled with the disjointness invaraint, Spanner also maintains the monotinicity invariant which states that within each Paxos group, timestamps to Paxos writes are assigned in monotonically increasing order.

Finally, Spanner maintains the external-consistency invariant which states that if a transaction T_2 starts *after* the commit of a transaction T_1, then the commit timestamp of T_2 must be *after* the commit timestamp of T_1. This invariant is maintained with the help of the following steps. Let us denote by e_i^{start} and e_i^{commit} the start and commit events of transacion T_i. Let s_i denote the commit timestamp of transaction T_i. The invariant we seek can be written in symbols as $t_{abs}(e_1commit) < t_{abs}(e_2^{start}) \rightarrow s_1 < s_2$. This condition is met by Spanner via the following two rules that govern the execution of transactions and assigning timestamps to transactions.

Let e_i^{server} denote the arrival event of the commit request at the coordinator leader for a write T_i.

- Start Rule: The coordinator leader for transaction T_i assigns a commit timestamp s_i no less than the value of $TT.now().latest$, computed after e_i^{server}.

- Commit Wait Rule: The coordinator leader ensures that clients cannot see any data written by T_i until time is past s_i, which in other words means, until $TT.after(s_i)$ is true. This rule ensures that $s_i < t_{abs}(e_i^{commit})$.

Using the above two rules, we can now *prove* that Spanner successfully maintains the invariant $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \rightarrow s_1 < s_2$ via the following inequalities.

By hypothesis, $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start})$. Due to commit wait, $s_1 < t_{abs}(e_1^{commit})$. Causality of events indicates that $t_{abs}(e_2^{start}) \leq t_{abs}(e_2^{server})$. By the start rule above, we also have that $t_{abs}(e_2^{server}) \leq s_2$. Putting together the chain of inequalities, we get that $s_1 < s_2$.

Snapshot Reads and Snapshot Transactions Alike the details mentioned for Read-Write transactions, Spanner along with Truetime provides simple and efficient mechanisms for supporting snapshot reads and snapshot transactions. Some of the details follow. For a more elaborate treatment, we refer the reader to [29].

For a snapshot read, there are two distinct steps. In the first step, for transaction T, spanner has to assign a timestamp T_{read}, and in the second step, the read has to executed at time T_{read} and result sent to the client. The big question to satisfy the consistency guarantees is to find a good value for T_{read}. While the latest time, given by $TT.now().latest$ is a possibiliity, this choice can introduce some inconsistencies. In particular, some of the reads with the above choice may have to block as we will explain shortly. Hence, spanner uses the following technique.

Recall the monotonicity invariant that spanner ensures at all times. We argue that using this invariant, it is possible to infer whether the state of a replica is sufficiently up-to-date to serve reads at a given timestamp. To this end, each replica maintains a variable, t_{safe}, that corresponds to the latest time at which it is up-to-date. Using t_{safe}, it holds that a replica can serve a read at timestamp t if $t \leq t_{safe}$.

We now discuss how a replica can obtain t_{safe}. Notice that there are two aspects in obtaining t_{safe}. One set of changes to the state of the replica are driven by the Paxos group and another set of changes are driven by the transaction manager. Let us denote the safe time for both of these as t_{safe}^{Paxos} and t_{safe}^{TM}. Then, $t_safe := \min\{t_{safe}^{Paxos}, t_{safe}^{TM}\}$. We now see how to obtain these safe times with respect to Paxos and the TM.

Note that t_{safe}^{Paxos} can be set to be the largest time at which Paxos write is done. Since Paxos write timestamps are assigned in monotonically increas-

ing order, no writes are pending with a timestamp that is at or before the above choice of t_{safe}^{Paxos}.

For obtaining t_{safe}^{TM}, spanner again relies on the idea that Truetime allows any replica acting either as a participant leader or a participant slave in a 2-phase commit can infer a safe timestamp as follows. Each replica records the time $s_{i,g}^{prepare}$ corresponding to a lower bound time of the prepared to commit record for a transaction according to the 2-phase commit protocol. Recall further that the commit timestamp of any transaction, s_i, is given such that $s_i \geq \max_g s_{i,g}^{prepare}$ over all replica groups that are participants in the transaction commit protocol. Thus, any replica in any group g can set $t_{safe}^{TM} := \min_i(s_{i,g}^{prepare}) - 1$ over all transactions T_i prepared at group g.

Let us come back to the question of why $TT.now.latest()$ is not a safe time for a snapsht to be read. We mentioned that using the above timestamp may results in reads blocking. This happens in particular if t_{safe} as described above has not advanced to sufficiently. Not blocking such a read with the above timestamp can lead to inconsistencies. Therefore, spanner uses a timestamp t_{safe} that is the latest time that allows the spanner system to be externally consistent even for snapshot read transactions.

Summary of Spanner Even as spanner was developed as a database that stores ⟨key,value⟩ mappings, the usage and adoption of spanner made it evolve into a fully relational database system with support for Standard SQL. Bacon et al. [11] describe this evolution and support for SQL.

9.7 Chapter Summary

In this chapter, we covered a wide set of topics that are practically significant in the context of distributed systems. The notion of transactions and distributed commit protocols are essential to many real-world applications inlcuding financial transactions. Skeen [109] develops a state-diagram view of the commit protcols that offer a view of how the protocols work and the states of the coordinator and the participants at various stages of the protocol.

We then studied how some of the strong consistency requirements and guarantees of transaction processing can be simplified in certain settings. This led to the study of what are called as no-sql databases. We then studied some example no-sql database systems such as Bigtable and Dynamo where the goal is to allow for efficient access in a distributed setting and increasing the availability of the system despite network induced partitions. Both of

these share some similarities that are important from a distributed systems perspective. For instance, both rely on using commodity hardware for ease of use and flexible scaling. The failures that ensue due to such commodity hardware are handled by appropriate fault-tolerance mechanisms. Most client operations using these systems avoid the single point-of-failure which is ensured by not having a single master that has to play a role in each operation. By distributing the operation across nodes, these systems do not need all operations to converge on any *central* master node.

There exist open-source version of no-sql databases such as Cassandra from Apache [23]. These systems however do not support a full set of functionality such as transactions spanning multiple tables. Finally, we studied Spanner, which Google refers to as a planet scale database that can (almost) ensure transactions with strong ACID semantics. This is made possible with the help of Truetime (see also Section **??**) that provides a strong NTP like time service at planet scale.

Beyond the ones discussed in the chapter, there are other distributed database systems such as Megastore [12], Scatter [52], VoltDB [115], Calvin [114] and the like. Calvin [114] promises to offer deterministic, ordered, ACID-compliant, execution of distributed transactions. These features allow for scalability. However, Calvin suffers one limitation that transactions whose read/write footprint is not statically determined require a recee model to execute. In other words, the authors of Calvin suggest that if the read-write footprint of transaction is not statically determined, then a trial run of the transaction be performed to know the required footprint so as to eventually execute the transaction knowing the full read-write footprint. Of course, the values that the first recee execution reads may change in the interim and this warrants a reexecution of the transaction. While this solution is practical, and does not impact performance in most cases, this solution nevertheless suffers from completeness.

Questions

1.

Chapter 10

Blockchains

Chapter 11

Distributed Machine Learning

Part III

Management

Chapter 12

Middleware

Job Management

Chapter 13

Fog and Edge Computing

Chapter 14

Distributed Programming Frameworks

Recall that distributed systems are characterized as a collection of autonomous computers communicating over a network of links. Some typical features of such a system are:

- There is no common physical clock and clocks can drift. Systems can be asynchronous too.
- The set of machines do not have common shared memory. This requires them to use messages for communication along with their semantics.
- The systems can be widely separated in a geographical sense. This allows them to be used in a variety of application such as SETI [4]. However, this comes with the challenge of programs having to deal with large message latencies.
- Nodes in the system are loosely coupled but cooperate with each other to achieve a common goal.
- Systems are naturally heterogeneous in their capabilities and functionalities. In such cases, it may not be possible to run some computation on some machines.

For the above reasons, distributed programming is inherently challenging. In essence, one needs to ensure that several geographically separate computers collaborate efficiently, reliably, transparently, and in a scalable manner. This idea leads to the notion of inherent complexity and accidental complexity as Brooks [21] mentions in the context of software systems.

Inherent Complexity: Inherent complexity refers to aspects of distributed programming that are inherent to the system model. Inherent complexity arises due to factors such as communication latency in the system, failures and recovery from failures of individual or groups of nodes in the system, and lack of physical clock which induces difficulties in arriving at an ordering of the events. Other factors that introduce inherent complexity in distributed programming include the possibility that failures in the system can introduce partitions that render unable communication across the system. Finally, there exist multiple aspects of security that the application has to contend with.

Accidental Complexity: Accidental complexity refers to aspects of distributed programming that can be addressed with appropriate technological interventions and enhancements. Accidental complexity arises due to using low level APIs that do not provide a good level of abstraction, lack of appropriate support for debugging and fault localization, and improper choice of algorithmic techniques. In addition, various components of distributed systems evolve in different trajectories and distributed programms have to contend with changes to key components in the lifecycle of the distributed program.

14.1 Common Ideas Across Distributed Programming Frameworks

In this section, we try to highlight common features across most distributed programming frameworks.

14.1.1 Client-Server Vs. Peer Model

Distributed programming framework usually support two models. In the Client-Server model, the client application and the server program usually reside on independent machines and the programming framework allows for interaction between the client and the server and provides the necessary support for such interaction. Viewed differently, client request a certain service from the server and the server responds with the service. In the peer model, nodes participate as peers and there is no designated set of machines labelled as clients or servers.

Examples of the former include Remote Procedure Call (RPC), gRPC, GraphQL, Thrift, Kafka, and the like. In the latter, we have frameworks

such as MPI, Erlang?, Map-Reduce?,

14.1.2 Synchronous Vs Asynchronous

One can also study distributed programming frameworks in terms of their support for synchronous communication only or also support for asynchronous communication. In synchronous communication, the sender and the receiver are waiting for the transmission and receipt of data. With asynchronous communication, the sender and the receiver do not have such a-priori arrangement. There is an additional level of detail with respect to message passing semantics. Some programming platforms support a blocking model where the sender an/or the receiver may be waiting for the send and the receive, respectively, to complete before moving to the next instruction in the program. In the non-blocking mode, the sender and receiver do not wait for the send and the receive to complete, respectively, before moving to the next instruction.

MPI offers support for both synchronous and asynchronous communication. More details of this are available in Section 14.2. In the synchronous case however, the programmer has to be careful to not introduce deadlocks in the code. (See Section **??** for more).

14.1.3 The Three P's

Just like parallel programming frameworks and languages, distributed programming frameworks also have to deal with striking a balance between the three P's of Portability, Performance, and Productivity. Portability refers to the ability of programs to work across different systems. Performance refers to the aspect of obtianing quality performance from a given program. Productivity refers to the ease with which programs can be expressed in the framework. Figure 14.1 shows the three P's along with distributed programming frameworks that satisfy these corners.

Distributed programming platforms such as MPI and gRPC are highly portable in general. On the other hand, platforms such as Map-Reduce offer high levels of prodcutivity.

14.1.4 Blocking Vs Non-blocking Communication

There are two semantic models for communication between nodes in a distributed system. In the blocking mode of communication, the node that issues the receive command blocks itself from further processing until the

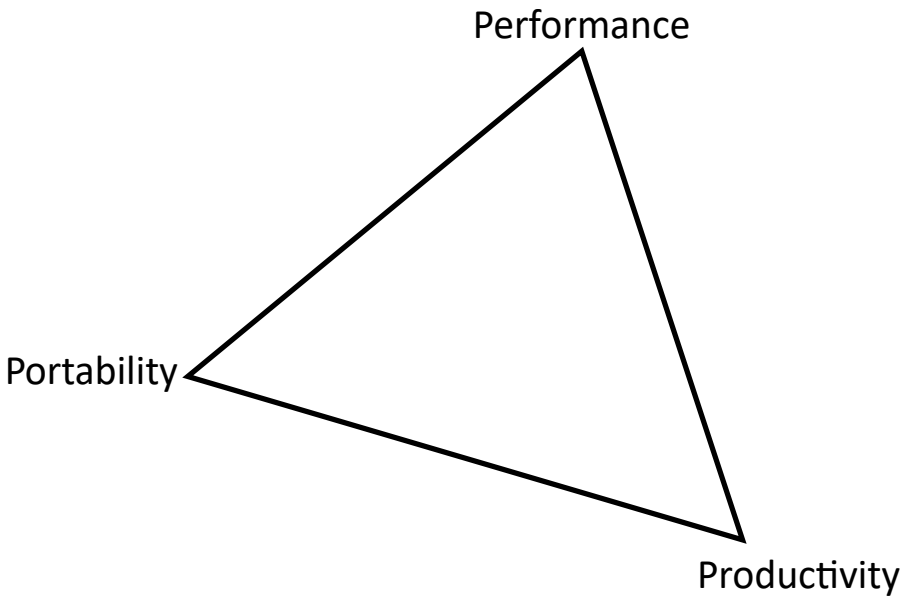


Figure 14.1: The three corners of distributed programming farmeworks.

message is received from the intended receiver of the message. In the non-blocking mode, the node does not wait for the data transfer to actually happen. In particular, in a non-blocking receive operation, the node moves to the next instruction even if the data may not have arrived from the intended sender.

Both these modes have their particular use cases, especially in distributed programming. Various distributed programming frameworks support either or both of these communication modes. Indeed, it is possible to also talk about communication modes such as synchronous blocking, synchronous non-blocking, asynchronous blocking communication.

14.1.5 Common Primitives

Distributed programming frameworks usually support a set of common primitives for communication mechanisms such as send() and receive(), synchronization constructs such as wait(), and in some cases collective operations that work on data at every node.

To summarize, we show Table **??** that shows the support that various current distributed computing platforms provide.

While the three paltforms we describe in this chapter are reprsentative,

Characteristic	MPI	gRPC	Map-Reduce
Synchronous			
Asynchronous			
Blokcing			
Nonblocking			
C-S			
P-P			

there exist other platforms too. Some of them such as Erlang are application specific, some such as Hadoop, GraphQL, are open-source versions of the above, and some such as CORBA, Java RMI, are earlier avatars of the ones we describe in the following.

14.2 MPI

Message Passing Interface (MPI) is a specification for a message passing library. This does not refer to a product or an implementation and is not specific to any compiler or programming language. The specification is defined as a standard and posted at www.mpi-forum.org. Since its definition, there have been numerous text books and other online material describing the MPI framework and its application to several problems. In this section, we provide a brief summary of MPI along with an example program.

MPI builds on the notion of a process and provides mechanisms for communication among processes running on parallel computers, clusters, or heterogeneous networks. MPI is a portable framework and is a useful way to provide for libraries that eventually obviates the need for every user to be an expert at writing MPI programs.

MPI combines interprocess communnication with syncrhonization. In general, consider just two processes, Process P_0 and P_1. If P_0 sends some data intended for P_1, this is done by an explicit send operation. The receiving process, P_1, also blocks when expecting data from P_0. Processing at P_1 resumes only after receiving data from P_0. Note however that MPI-2 allows a divergence from this semantics and decouples communication and synchronization. These are called as one-sided operations.

MPI supports a peer model of interprocess communciation. Each participating process gets a unique identifier denoted `rank`. A process can obtain its rank by using the MPI function `MPI_COMM_RANK()`. This function returns a number between 0 and the one less than the number of processes that identifies the rank of the process calling the function. Any participat-

ing process can know the number of participating processes via the MPI function `MPI_COMM_SIZE()`. This function returns the number of processes.

Using the above two functions, one can already write a simple MPI program as shown in Listing 14.1.

Listing 14.1: A first Simple MPI Program

```
1  #include <stdio.h>
2  #include <mpi.h>
3  int main( int argc, char *argv[] )
4  {
5      int myRank, total;
6      MPI_Init( &argc, &argv );
7      MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
8      MPI_Comm_size( MPI_COMM_WORLD, &total);
9      printf("My rank is %d out of %d processes.\n",
10       rank, size );
11     MPI_Finalize();
12     return 0;
13 }
```

The program produces an output that has one print statement per each participating process. In the program above, the parameter MPI_COMM_WORLD refers to the default group to which all participating processes belong to. Processes can be organized into more groups based on application specific requirements. For instance, in a program that operates on a matrix input, all the processes that work on elements in a row can be made as part of a group. The function MPI_COMM_SPLIT creates a new group. To identify a new group, the MPI_COMM_SPLIT function takes in a paraemeter passed by reference and the value of this parameter is a way to identify the new group. As an example, suppose the default group MPI_COMM_WORLD has 16 processors and we want to create a subgroup that contians processors that have an even rank. The split command to achieve this is MPI_SPLIT(MPI_COMM_WORLD, rank/2, rank, &evenGroup) where rank contains the rank of the processor in the default group. This rank can be obtained using the MPI_Comm_rank function as in Listing 14.1.

In addition to groups, MPI uses another concept called the context. Messages are sent and received in a specific context. A context and a group together form a communicator. Each process participating in a group has a rank within that group. In the earlier program listed in Listing 14.1, the ranks are between 0 and one less than the number of processes since all processes belong to the default group MPI_COMM_WORLD.

In our discussion and examples in the following we will restrict ourselves to using a single group, the default group.

MPI Send

We move to describe the semantics and syntax of the MPI send function. The function MPI_SEND takes six paramters as input and has the following syntax.

```
MPI_SEND(start, count, datatype, destination, tag, group)
```

The first three parameters, start, count, and datatype, define the message buffer. The fourth parameter destination refers to the number of the receiving process in the communication group corresponding to the sixth parameter. The fifth parameter, tag, helps the receiver in identifying the message. This is useful in a context where one process sends multiple messages to a receiver and with possible message reordering, the reciever needs additional information to contextualize the received messages. In order to simplify the usage, MPI allows a default tag MPI_ANY_TAG.

Note that tags are different from datatypes. Data types allow for applications using MPI to specify the type of the data. This is useful when the machines on which the processes execute have varying formats and representations for various data types. Datatypes therefore help MPI support heterogeneity naturally.

MPI Receive

We now describe the semantics and syntax of the MPI receive operation. The receive operation, MPI_RECV takes seven parameters as input has the following syntax.

```
MPI_RECV(start, count, datatype, source, tag, comm, status)
```

A process executing the MPI_RECV function waits until a matching (on source and tag) message is received from the system and the buffer can be used. The match is applicable to both the specified source and the tag, the fourth and the fifth parameters of the function. The source is specified as the rank of the sender process in the communictor `comm`. If no specific sender is intended, then MPI_ANY_SOURCE is to be specified as the fourth paramter.

The message is copied to the buffer specified by the first three parameters start, count, and datatype. Receiving fewer than count occurrences of datatype is not an error but receiving more is an error.

The last parameter `status` contains information on the status of the recieve operation.

Listing 14.2 show a small extension to Listing 14.1 by having the process with rank 0 to send a number to all the other processes. The other processes receive the number and print the number they receive.

Listing 14.2: Using MPI_Send and MPI_Recv

```
1  #include <stdio.h>
2  #include <mpi.h>
3  int main( int argc, char *argv[] )
4  {
5      int myRank, total, number;
6      MPI_Init( &argc, &argv );
7      MPI_Comm_rank( MPI_COMM_WORLD, &myRank );
8      MPI_Comm_size( MPI_COMM_WORLD, &total);
9  if (myRank == 0) {
10     for (i=1;i<total;i++) {
11         number = rand()%100;
12         MPI_Send(&number, 1, MPI_INT, i, 0,
               MPI_COMM_WORLD);
13     }
14 } else if (myRank != 0) {
15     MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
16              MPI_STATUS_IGNORE);
17     printf("Process %d received number %d from process 0\
           n",
18             myRank, number);
19    }
20     MPI_Finalize();
21     return 0;
22 }
```

14.2.1 Collectives

Recall that MPI structures send and recieve functions to a group of processes. Collective communication is another support MPI offers to applications where the communication is now over all processes in a given group. There are three different types of collective operations that MPI supports. These are for synchronization, collective computation, and data movement.

Collectives for Synchronization

Applications sometime need to ensure that all processes within a group reach a specific point in execution before *any* of them proceeds further. The MPI_BARRIER function supports such a model. The syntax for this operation is `MPI_BARRIER(group)`, where `group` is a communication group.

Processes in the specified group block until *all* processes in the group also call the function. This requires processes to wait at this function and forces a synchronization of all the processes in the group. While the use of such barriers is seen as not being conducive to parallel and distribtued programming, there are limited use cases where such barriers are useful.

Collectives for Data Movement

There exist several MPI functions that support movement of data over the process in a group. These come under the categories of one-to-all, all-to-one, and all-to-all communication.

In the first category, we have the broadcast and scatter operations. The broadcast operation, MPI_BCAST, has the following syntax and semantics.

```
1 MPI_BCAST(start, count, datatype, root, comm)
```

The first three parameters are as in the MPI_SEND function. The fourth parameter is the process that acts as the source of the broadcast. The message is sent to all processes in the group specified by `comm`.

Another operation that MPI supports in the first category is the scatter operation. The semantics of the scatter operation in general is to distribute a set of data items to a set of processes. The syntax of this function is as follows.

```
1 MPI_SCATTER(sData, sCount, sDatatype, rData, rCount,
     rDatatype,
2            root, comm)
```

The first parameter indicates the source of the data to be sent. The second parameter specifies the number of data items of type sDatatype that is to be delivered to each of the processes in the communication group specifed by the last parameter `comm`. The parameters prefixed with "r" refer to the buffer, number and datatype at the destination process. From sData, sCount data items of the specified type are moved to each of the processes in the group. The first sCount items go to the process with rank 1 (in the group), the next sCount items go to the process with rank 2 and so on. The parameter `root` indicates the rank of the source process.

We now move to the second category of collective data movement operations. The all-to-one operations in MPI include the gather operation. The gather operation is in a way has the opposite semantics to that of the scatter operation. Just as the scatter operation works to move elements from a designated source to all other processes in the group, the gather operation achieves the opposite effect. Data from all other processes in the group is moved to a designated source process. In addition, the operation orders elements according to the rank of the process from which they were received.

The syntax of the gather operation, MPI_GATHER, is as follows.

```
1 MPI_Gather(sData, sCount, sDatatype, rData, rCount,
     rDatatype,
2            root, comm)
```

The first three parameters indicate the buffers at each process that contain the dataitems being transfered. The parameter `root` is the destination process. For this process, the parameters rData must be a valid buffer and the other processes can send a NULL filed for this parameter. Another detail that is important to note is that the field rCount is the number of dataitems received from each process and not the total number of dataitems received.

The final category of collective data communication operations include the all-to-all category. There are two operations in this category: MPI_ALLGATHER and MPI_ALLTOALL.

The first of these transfers data from each process the following

The second of these works in the following setting. Each process in a communication group has data. This data at each process is to be sent to all other processes. While this communication can be achieved by means of individual broadcasts, there is scope for an efficient implementation of the All-to-All communication especially when the data involved is small in size. The syntax of this function is as follows.

```
1 int MPI_Alltoall(sData, sCount, sDatatype, rData, rCount,
      recvDatatype, comm)
```

The first three parameters correspond to the data buffer, the number of dataitems, and the type of the data being transfered. The next three parameters prefix "r" correspond to similar ones at each process. The last parameter refers to the communication group.

14.2.2 Collectives for Computation

We now describe the collectives that peform a computation based on the data across the processes in a communication group. Prominent among these are the Scan and Reduce operations. These terms are popular in the parallel computing parlance and have the following meaning. A reduce operation applies an associative operation on data items from each process and produces one output. A scan operation applies an associative operation on data items from each process and produces one output at each process corresponding to applying the function data items at processes of ranks at most the process. REWORD.

The syntax for the MPI scan operation is as follows.

```
1 MPI_SCAN(sendbuf, recvbuf, int count, datatype,  op, comm
    )
```

In the MPI_SCAN function, the parameters sendbuf and recvbuf correspond to the input dataitems and the space for the output data item. The parameters count and datatype correspond to the number of dataitems and the type of the data. The parameter `op` refers to the operation to be applied. MPI has inbuilt support for standard operations such as addition, maximum/minimum, product, bitwise and/or, logical and/or, and the like. These are usually specified as MPI_SUM, MPI_MAX, and so on. The last parameter `comm` refers to the communication group on which the function is applied. As an example, if the group has four processes with values 2, -2, 1, and 4, and the operation is MPI_SUM, the result of the scan operation is 2, 0, 1, and 5.

The reduce operation, MPI_REDUCE has the following syntax.

```
1 MPI_REDUCE(sDdata, rData, count, datatype, op, root, comm
    )
```

The parameters have meanings similar to that of the MPI_SCAN function. One difference is that the result is stored in the process with rank specified by `root`. The other processes can send NULL for the parameter `rData`. The function is applied to the processes in the group specified by `comm`. The parameter `op` refers to the operation.

To summarize, in Table **??** we provide the output of the various collective operations in the setting where there are four processes holding a value each.

Process	P_1	P_2	P_3	P_4
Broadcast				
Initial Value	4	3	1	2
Received Value	3	3	3	3
Scatter, root = 2, sCount= 2				
Initial Value		[3, 8, 6, 2, 1, 5, 7, 4]		
Received Value	[3, 8]	[6, 2]	[1, 5]	[7, 4]
Gather, root = 1, rCount=1				
Initial Value	4	3	1	2
Received Value	[4, 3, 1, 2]			
AllGather, root = 1, rCount=1				
Initial Value	4	3	1	2
Received Value	[4, 3, 1, 2]			
All-to-All				
Initial Value	4	3	1	2
Received Value	[4, 3, 1, 2]	[4, 3, 1, 2]	[4, 3, 1, 2]	[4, 3, 1, 2]
Scan, op=+				
Initial Value	4	3	1	2
Received Value	4	7	8	10
Reduce, root=4, op = MAX				
Initial Value	4	3	1	2
Received Value				3

Table 14.1: Table shows an example of various MPI collective operations.

14.2.3 A Simple MPI Program

Armed with the simple toolkit of the MPI functions, we now proceed to develop a simple MPI program. We consider the computation of π using the idea that the number of points that fall inside a unit circle that is inscribed in a unit sqaure approaches $\pi/4$. We develop this program by having each process do the random experiment of picking a set of points and counting how many of these points fall inside the unit circle. The overall average is then used to approximate the value of π.

In this direction, let N be the number of samples we use and n be the number of processes. Each process draws N/n samples and uses the variable ni to denote the number of points that fall inside the circle.

Listing 14.3 shows the entire program.

Listing 14.3: Computing π using MPI

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char *argv[])
5  {
6      int done = 0, N, myid, n, ni, i, rc;
7      double PI = 3.14159265358979323846264 3;
8      double mypi, pi, h, sum, x, a;
9
10     MPI_Init(&argc,&argv);
11     MPI_Comm_size(MPI_COMM_WORLD,&n);
12     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
13     while (!done)  {
14         if (myid == 0) {
15            printf("Enter the total number of samples
16                        (N): (0 quits) ");
17            scanf("%d",&n);
18            }
19         MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
20         if (N == 0) break;
21         ni   = N / n;
22         sum = 0.0; sumi = 0;
23         for (samples = 0; samples < ni; samples++) {
24             // draw a point in the unit square
25             x = (double)rand()/RAND\_MAX;
26             y = (double)rand()/RAND\_MAX;
27             //check if the point lies inside
28             //the unit circle
29             if (x*x + y*y <= 1) sumi ++;
```

```
30
31          }
32          pii = 4*sumi/ni;
33          MPI_Reduce(&pii, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
34                  MPI_COMM_WORLD);
35           if (myid == 0)
36           printf("pi is approximately %.16f, Error is %.16
                f\n", (double)pi/n, fabs(pi - PI25DT));}
37           MPI_Finalize();
38          return 0;
39  }
```

Building and Running the Program

Unlike standalone programs, the steps to follow to be able to run an MPI programs on a cluster is usually somewhat dependent on the installation. If you are running MPI on your multi-core machine, it is a bit straight-forward. We will only mention the steps needed to run an MPI program on a single multi-core machine. For the other case, it is best to discuss with your local cluster/facility administrator.

On your multicore machine/laptop, you need to first install the MPI package such as mpich or OpenMPI. This installation procedure depends on the OS that your machine runs on. For typical Linux based machines, you can use apt-get or yum to do the job. After this installation is complete, you use `mpicc` to compile MPI programs. You can use simple `make` scripts to set up a smooth compilation workflow. Once the executable is ready, you use `mpirun` to run the MPI program. The syntax is `mpirun -np` <num> <file>. In the above, <num> refers to the number of cores you want your MPI program to use and <file> is the name of the executable file.

Some Pitfalls

MPI supports synchronous and blocking send and receive operations. While most programs expect such a model, it is possible that deadlocks can be created due to these operations. Consider two MPI processes, P_1 and P_2, with P_1 waiting to receive data from P_2 post which it sends data to P_2, while at the same time, P_2 wants to receive data from P_1 before it sends data to P_1. Listing **??** shows the corresponding code snippets.

The trouble with the above listing is that P_1 waits to recieve a message from P_2 and vice-versa. There exist several ways to avoid this and other such pitfalls. In this case, one can reorder the send and receive instructions

```
1 $\vdots$                1 $\vdots$
2 MPI\_recv(2, &data, )2 MPI\_recv(1, reply, )
3 if (true) MPI\_send(23 if (reply) MPI\_send
      reply, )              (1, &data, )
4 $\vdots$                4 $\vdots$
```

at either P_1 or P_2 to remove the circular waiting. Another way to avoid this particular danger is to use asynchronous send and recieve operations supported by MPI2. These are the iSend() and iRecv() operations and we leave it to the reader to explore the semantics of the asynchronous operations.

14.2.4 MPI Summary

MPI has been found to be quite useful in writing distributed programs and has a strong userbase. There are some pitfalls one has to be aware of. In particular, the send and receive operations in MPI are blocking in their semantics. This can cause deadlocks. For instance, if the receiving process does not have enough space, then the sender waits for this space to be made avaialble.

MPI2 offers support for non-blocking variants of the send and receive operations. These can avoid deadlocks but introduce other issues that the programmer has to be aware of.

MPI is a useful tool to create libraries, however one drawback is that MPI does not offer fault tolerance.

14.3 Map-Reduce

As we note from Section 14.2, MPI does not offer any fault tolerance. One of the possible explainations for this lack of support is that MPI often is used on server scale infrastructure that comes with fault tolerance. However, maintaining server scale infrastructure is often expensive. On the other hand, commodity hardware is ubiquitous and cheap but comes at the expense of reliability. One challenge therefore is to see if large-scale distributed programming can be supported on commidty hardware with additional support for fault tolerance.

Another lacunae of MPI is the lack of emphasis on productivity. MPI programmers have to deal with a low level of programming abstraction. Given the complexity and usage patterns of distributed programming, a

high level of abstraction can help programmers achieve productivity without impacting performance.

These considerations led to the development of the distributed programming framework, Map-Reduce, by Dean et al. in 2005 [34]. The Map-Reduce framework runs on commodity hardware that are usually available in bulk and are less expensive. The framework has to however work with failures to machines or components of machines and also contend with low network and disk bandwidth.

Ever since the release of Map-Reduce, there have been several similar frameworks. Apache released Hadoop , which is an open-source version of Map-Reduce. Amazon has its own similar offering called PaaS, standing for Platform-as-a-Service. In the following, we explain the Map-Reduce programming framework along with a complete example.

14.3.1 The Map-Reduce Programming Model

The Map-Reduce programming model envisages expressing distributed programs via two functions: a mapper and a reducer. It is possible to express the solution to many problems from domains such as image processing, graph computations, and machine-learning algorithms as a sequence of mapper and reducer functions. While the model is not general-purpose and cannot express all computations, the versatality of the map-reduce model is in its ability to express several computations in the map-reduce model. REWORD–NOT SMOOTH TEXT

The Map Function The map function takes as input a list of key, value pairs. The output of the function is a list of key, value pairs. The map function produces zero or more values per each input key. The Map-Reduce framework supports certain in-built functions that can be applied during the mapper. Examples of such functions include toUpper() that converts a string argument to a string consisting of all uppercase letters.

The Reduce Function The Reduce function

A Simple Example Let us imagine that the Aadhar data is stored as a sequence of records, one per each line, with various fields such as aadhar number, address pin code, first name, last name, year of birth, address, last update date, among other fields. Figure **??** shows an example set of records.

Suppose we want to find the most common first name by pin code. We treat each line of the input as a key,value pair and use the mapper function

1, 1093 6233 8742, 100205, vivek, rao, 1997, Apt 23 Krishna Layout Chennai, 2021-08-23-161523
2, 8372 7362 6532, 203303, sara, jacob, 2011, 1-3-231/A Roy Enclave Kolkata, 2023-05-16-100245
3, 3293 7462 8730, 650237, srinivas, krishna, 1986, House no 236 Gandhi colony Vijayawada, 2015-04-22-1435
⋮

⟨132980, [rama, vikas, rama, sai]⟩
⟨225682, [madhu, ankit, krishna]⟩
⟨763210, [vivek, sara]⟩
⟨132980, [kapil, charan, rama]⟩
⋮

to emit output key,value pairs of the form ⟨ pin code, first name ⟩. A sample set of such pairs look as follows.

⟨132980, rama⟩⟨225682, madhu⟩⟨763210, sara⟩⟨132980, rama⟩⋮

Now, the reduce function gets as input key-value pairs that have a list of values per each key. The key is the pin code and the list of values contains all the first names of people in that pin code as Figure **??** shows.

The reduce function can pick the most frequent name per each pin code and produce tuples of form ⟨pin code, frequent-first-name⟩.

Shuffle and Sort To understand how the output from a mapper is transformed as input to the reducer, we need to delve a little deeper into the Map-Reduce framework. In addition to the map and the reduce function, which are user-defined, the real support that the Map-Reduce framework provides is via its in-built functions that shuffle the output of mapper functions and the sorting of keys done before calling the Reducer function.

The shuffle operation applies to the key-value pairs from all mapper machines. The key-value pairs are arranged into groups based on the keys. In other words, all the key-value pairs with the same key are grouped together. The framework applies these functions once *all* the mappers complete their task.

Before sending the (grouped) key-value pairs to reducer machines, the Map-Reduce platform does a sort of the keys. Sorting ensures that the keys at any reducer machine arrive in a sorted order. There is no sorting guarantee of keys across reducer tasks.

CHECK: what is sorted? Keys or values within a Key?

The Partitioner and Combiner From the above paragraphs, it appears that a program written in the Map-Reduce framework has four phases: mapper-shuffle-sort-reducer. Of these, the user has to implement the Mapper and the Reducer functions while the framework has efficient support for the shuffle and sort functions. There are two other optional functions that the framework provides. These optional functions provide further opportuniteis to optimize the program.

The combiner function applies to the output of each mapper. The combiner runs on the mapper machines and the key,value pairs output and applies a reduce function on these key,value pairs. The framework however may call the combiner function zero, one, or more than one time. This means that the functions that the programmer can use as part of the combiner routine should be such that the output is unaffected by its repeated application. The real benefit of the combiner is to act as a mini-reducer thereby decreasing the volume of data that is carried over to the shuffle and the sort stages. However, the combiner cannot replace the role of the Reducer since the combiner function is applied only on the data local to the mapper.

The partitioner function allows the programmer a limited control on the reducer machine that key,value pairs from the shuffle and the sort stage go to. By default, the framework sends each key,value pair to a reducer machine chosen uniformly at random based on the key. This default partitioner, also called as the HashPartitioner, uses a hash function on keys to map each key to a reducer. The hash function that the framework uses is chosen so as to balance the distribution of keys to the reducer machines. Notice that balacing with respect to the number of keys is no guarantee on the number of values processed by a reducer.

The Map-Reduce Nomenclature

Datanode

namenode

14.3.2 A Complete Example: Histogram

Listing 14.4: Compuing a histogram using Map-Reduce

```
1  int bucketWidth = 8 // input
2
3  Map(k, v) {
4      emit(floor(v/bucketWidth), 1)
5      // <bucketID, 1>
6  }
7
8
9
10 // one reduce per bucketID
11 Reduce(k, v[]){
12     sum=0;
13     foreach(n in v[])  sum++;
14     emit(k, sum)
15     // <bucketID, frequency>
16 }
```

Building and Running the Program Based on the installation, there are minor challenges in how one can bulid and execute the program.

14.3.3 Other Features

Recall that Map-Reduce offers fault-tolerance so as to accommodate potential failures to machines or components of machines. Map-Reduce scales to run on thousands or more of commodity machines and hence failures are common. One way of providing for fault-tolerance is via the JobTracker and the TaskTracker. Note that while retaining the functionality, the second version of Map-Reduce calls these as ResourceManager and ApplicationMaster.

The JobTracker receives requests for execuitng MapReduce pograms from the client. The JobTracker works in coordination with a TaskTracker in identifying and addressing failures. JobTracker assigns each MapReduce job to a TaskTracker based on parameters such as locality of the input data.

14.3.4 Other Variants

Map-Reduce is a product that came out of Google [34]. Recall that the Map-Reduce framework runs using the Google File System (GFS) as the

underlying data store. Apache released an open-source version of Map-Reduce, Hadoop [5]. Hadoop runs using the Hadoop Distributed File System (HDFS), an Apache open-source variant of GFS, as the underlying data store.

YARN, standing for Yet Another Resource Negotiator, is another varaint of Map-Reduce. In fact, YARN addresses some of the scalability concerns with Map-Reduce and can scale up beyond 10000 nodes. To this end, YARN has more sophisticated fault tolerance mechanims. Recall from Section **??** the failure of the JobTracker endangers the Map-Reduce applications currently under execution. The ResourceManager of YARN handles this failure by saving enough state information.

14.3.5 The Map-Reduce and Other Related Technologies

We now discuss how the Map-Reduce platform differs from other related technolgoies.

Scripting Languages such as `awk` and `grep` Some of the mapper functionality and the reducer functionality can be implemented using sophisitcated Unix style scripting languages such as `awk` and `grep`. Other high-level abstractions such as PERL and Python also provide a strong suite of libraries and routines to support typical mapper and reducer functionality.

However, the true power of Map-Reduce is in being able to offer an abstraction for parallel processing of large data. Tools such as `awk` and `grep` do not offer support for parallel processing. High level languages such as Perl and Python offer linkages to parallel and distributed programming frameworks but do not cater to the kind of abstraction that Map-Reduce can provide in particular for parallel processing.

Map-Reduce Vs. RDBMS It appears that the functionality that Map-Reduce supports can be achieved via relational database systems and their query languages. However, there are key difference between these two technologies.

- The scale of the data that Map-Reduce applies to can be an order of magnitude bigger than what a typical relational database can store. Map-Reduce is indeed aimed at processing big data.
- It appears that Map-Reduce would scan each record and apply a user function on the record. This can be supported easily by a query to

an RDBMS also. However, the read model of RDBMS is usually optimized for point queries. RDMBS also have a huge read latency and some of this latency is offest by using indexing. There is no such need to support indexing on data that is provided as input to the Map-Reduce framework.

- RDBMS works well with strucutred data. Most RDBMS table designs also perform normalization whose main goal is to remove redundant storage and provide for data integrity. One drawback of normalization is that a complete record read has to be translated to a set of non-local reads. On the other hand, Map-Reduce assumes that it is possible to perform high-speed streaming reads and writes.

- An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data.

However, the two technologies namely RDBMS and Map-Reduce are seeing a convergence of sorts. There exist RDBMS systems such as Aster Data and Greenplum that incorporate certain aspects of Map-Reduce in their support. Similarly, language abstractions such as those supported by Pig and Hive built on top of Map-Reduce frameworks offer support for SQL style operations for users.

14.3.6 Summary

To summarize, the Map-Reduce framework is versatile to support a variety of computations involving large-scale data. The framework provides a high degree of abstraction and offers the user a transparent mechanism to deal with many issues of distributed programming such as scheduling and load balancing, data and task distribution, synchronization, handling faults, and so on. On the other hand, the programmer has to express the computation via only four operations: map, reduce, combiner, and partitioner.

14.4 gRPC

Remote Procedure Call (RPC) is a popular interprocess communication mechanism that has been in vogue for several decades. Nelson [18] introduced the earliest version of RPC in 1976. SUN put its own version of RPC in 1980 and used it its implementation of NFS [104]. RPC provides an

abstraction of allowing (client) programs to make function calls that get executed on a remote machine (server). The abstraction makes it transparent for the user to issue the function call. RPC itself internally uses an existing transport layer protocol such as UDP or TCP to prepare the function call, contact the server for executing the function, and report the results back to the client program.

Later versions of platforms that supported RPC included the middleware platforms such as Common Object Request Broker Architecture (CORBA) and Java RMI. These platforms added an object-oriented model to RPC and provided for data encapsulation and a separation of the method (function) interface and implementation. Google introduced gRPC [53] in the year 2015 and is based out of its own in-house RPC framework Stubby that allowed multiple microservices within the Google datacenters to connect and communicate with each other. One trouble with Stubby is its strong coupling to Google internal microservices thereby rendering it less generic. gRPC is a open source framework that is general-purpose and cross-platform ready while offering similar scalability, performance, and functionality to that of Stubby.

Many practices from the orignial definition of RPC and its later versions such as CORBA [56], SOAP [20], and REST [41] continue to be used in later versions such as gRPC. Some of these include the ideas about the Interface Definition Language (IDL), marshaling and demarshaling of parameters, synchronous communication based request-response model, platform neutrality, and the like.

gRPC offers several advantages over other RPC platforms such as CORBA. These are mentioned in the following.

- Use of HTTP/2: Unlike CORBA and Apache Thrift, gRPC uses HTTP/2 for its communication. HTTP/2 has inherent advantages such as using a single network connection and increases the efficiency of inter-process communication.

- Strong Typing: gRPC uses strong static typing thereby reducing the scope of errors in distributed programming.

- Multi-language Support: gRPC offers support for multiple languages to be used at both the server and the client end.

- Full Duplex Communication: gRPC provides in-built support for full duplex communication and hence naturally extends support for developing streaming server and streaming client based applications.

This communication model extends the simple synchronous request-response style of communication that existing RPC platforms are limited to.

- Additional Features: gRPC has native support for additional features such as authentication, resiliency, data compression, load balancing, and the like.

14.4.1 Using gRPC

In this section, we outline the steps needed to write a gRPC based client-server application. We imagine a hypethetical set up where a university offers a service based on the courses offered in an academic year such as the list of courses. Clients can request information about a particular course or a set of courses. We will extend this scenario with other services and mechanisms such as streaming, transcripts, and the like.

The gRPC Server

We first identify what the server will post as the functions that become part of the service provided by the server. In our example, the university service aims to support functions such as getting the course name, course syllabus, number of seats available, and the meeting schedule of a given course code. The course code is a seven character alpha-numeric string such as CS2.312. The course name and the course syllabus are of type string, and the number of seats available is a positive integer. We now write these functions in the Interface Definition Language.

```
1
2 rpc getCourseName(String courseCode)
3
4 rpc getCourseSyllabus(String courseCode)
5
6 rpc getCourseSeatsAvailable(String courseCode)
```

The gRPC Client

In the following, we show the client program that uses the above functions.

Building the Application

Listing 14.5: The Protocol Definition

```
syntax = "proto3";

option java_package = "ex.grpc";

package CourseInfo;

// The course information service definition.
service CourseName {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

The client program looks as shown in Listing **??**. THIS IS A C++ PROGRAM. TO BE CHANGED TO JAVA

Listing 14.6: gRPC Client Example

```
#include <iostream>
#include <memory>
#include <string>
#include <grpc++/grpc++.h>
#include "helloworld.grpc.pb.h"
using grpc::Channel;
using grpc::ClientContext;
using grpc::Status;
using helloworld::HelloRequest;
using helloworld::HelloReply;
using helloworld::Greeter;
class GreeterClient {
 public:
  GreeterClient(std::shared_ptr<Channel> channel)
      : stub_(Greeter::NewStub(channel)) {}
  // Assambles the client's payload, sends it and
     presents the response back
  // from the server.
```

```
18    std::string SayHello(const std::string& user) {
19      // Data we are sending to the server.
20      HelloRequest request;
21      request.set_name(user);
22      // Container for the data we expect from the server.
23      HelloReply reply;
24      // Context for the client. It could be used to convey
            extra information to
25      // the server and/or tweak certain RPC behaviors.
26      ClientContext context;
27      // The actual RPC.
28      Status status = stub_->SayHello(&context, request, &
            reply);
29      // Act upon its status.
30      if (status.ok()) {
31        return reply.message();
32      } else {
33        std::cout << status.error_code() << ": " << status.
              error_message()
34                  << std::endl;
35        return "RPC failed";
36      }
37    }
38   private:
39    std::unique_ptr<Greeter::Stub> stub_;
40  };
41  int main(int argc, char** argv) {
42    // Instantiate the client. It requires a channel, out
          of which the actual RPCs
43    // are created. This channel models a connection to an
          endpoint (in this case,
44    // localhost at port 50051). We indicate that the
          channel isn't authenticated
45    // (use of InsecureChannelCredentials()).
46    GreeterClient greeter(grpc::CreateChannel(
47        "localhost:50051", grpc::InsecureChannelCredentials
              ()));
48    std::string user("world");
49    std::string reply = greeter.SayHello(user);
50    std::cout << "Greeter received: " << reply << std::endl
          ;
51    return 0;
52  }
```

The server program looks as shown in Listing 14.7

Listing 14.7: gRPC Server Example

```
1  void RunServer() {
2    std::string server_address("0.0.0.0:50051");
3    GreeterServiceImpl service;
4    ServerBuilder builder;
5    // Listen on the given address without any
       authentication mechanism.
6    builder.AddListeningPort(server_address, grpc::
       InsecureServerCredentials());
7    // Register "service" as the instance through which we'
       ll communicate with
8    // clients. In this case it corresponds to an *
       synchronous* service.
9    builder.RegisterService(&service);
10   // Finally assemble the server.
11   std::unique_ptr<Server> server(builder.BuildAndStart())
       ;
12   std::cout << "Server listening on " << server_address
       << std::endl;
13   // Wait for the server to shutdown. Note that some
       other thread must be
14   // responsible for shutting down the server for this
       call to ever return.
15   server->Wait();
16 }
17 int main(int argc, char** argv) {
18   RunServer();
19   return 0;
20 }
```

Installing and Executing a gRPC Program

Setting up gRPC requires a prior installation of the necessary compilers and other tools. One way to get this set up done on Linux based systems is to clone from the existing `git` repository at `https://github.com/grpc/grpc` ane executing the `make` followed by `make install` command that runs the setup scripts.

Once the installation is available,

Adding Streaming Support

Support for streaming client-server interaction is one feature that gRPC provides. In the streaming communication model, for one request from the

client, the RPC server can send multiple responses. These responses need not all be sent at once and can be sent as the server obtains more data that matches the response. The server marks the end of the stream by sending the status details of the server and trailing metadata to the client.

In terms of programming, the stream server model introduces a few differences. The return type of the server function that outputs a stream has to be given the type as a stream variable. At the client side, the program that calls such a stream output based remote service function has to process the multiple outputs until the end of the stream is reached.

In the gRPC framework, also the RPC client can operated in a streaming model. In this model, the client sends a sequence requests to the server. The server makes a single response to the entire sequence of requests. The server is not under an obligation to wait for all the requests to be received before sending its response. The server can send its response after reading one or more of the messages of the request sequence. The client program however includes a end of stream message to notify the server.

Applications that benefit from streaming request include those that involve large data transfers. The entire data can be chunked into more manageable sizes at the sending side. The receiving side recieves chunks of the data. One benefit in such cases it to reuse an established connection for transferring all the chunks instead of establishing new connections for each chunk. Another such use case is where the server may be interested in sending a reply based on data that is yet to arrive or yet to be processed. For instance, consider suggesting friends to a user on the facebook website. An initial set of, say a dozen, friends may be recommended at first. If the user continues to browse that functionality, the server may send another dozen recommendations. This allows the server to compute and send only when the response is in demand. In this usecase, it is not the size of the data that warrants a proper use of streaming, but it is the cost associated with preparing the reply that the server optimizes on.

gRPC also allows applications to use bidirectional streaming where both the client and the server send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like. For instance, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes. The order of messages in each stream is preserved.

Using Authentication

Like RPC, gRPC also provides optional features to enable authentication of the server, the client, and the communication channel between the client and the server. Transport Level Security (TLS) supports the commnication channel security in gRPC. A secure channel between the client and the server should offer support for a reliable and private connection between the client and the server. Reliability of the channel refers to the ability of the channel to support message integrity checking and to prevent alteration of data in transit in the channel. Private connection refers to the abiilty to use appropriate cryptographic mechanisms that establish a unique key for each connection. This unique key is usually set up via negotiation at the start of the connection.

gRPC supports a one-way or two-way secured connection. In the one-way mode, only the client validates the server to ensure that the messages received are indeed from the intended server. During connection establishment, the server shares its public certificate with the client. The client can verify the shared public certificate using a Certificate Authority (CA). Once the verification with the CA is successful, the client uses a shared key to send encrypted messages.

In the two-way mode, both the client and the server authenticate each other. Both the client and the server share their public certificate with each other. In turn, they approach the CA to validate the received certificate. Post the validation of the certificate by the CA, the client and the server use secure communication to contnue their exchange. The client initiates the flow of these events by contacting the server, obtianing the public certificate of the server, verifying the certificate with a CA, sending the certificate of the client to the server post-verifciation, the server verifying the certifciate of the client using a CA, and then enabling exchnage between the client and the server.

Beyond a secure channel, gRPC also provides for authenticating the identity of a client at the server side. gRPC supports three possible mechanisms to this end: the client can provide a username:password in Base64 encoded form, the client can use JSON tokens, and the client can use OAuth2 tokens. One advantage in using tokens is that tokes can be set with a time duration after which they expire.

Other Functionalities

gRPC provides support for lots of other functionalities. One such is interceptors. These are pieces of code that get invoked prior to the RPC call and act as useful mechanisms to perform any pre- or post-processing as part of the RPC call. Some functionality that one can push to such interceptor routines include logging, metrics, authentication, and the like. Reqeust interceptor is an interceptor that gets invoked during the request call and Response interceptor is an interceptor that gets invoked during the response call. Based on the nature of communication mechanism used by the client and the server, the program needs to implement and register the unary interceptor or the stream interceptor. gRPC allows for more than one interceptor function to be invoked as part of the RPC call. Example below shows the syntax of how to use an interceptor in the gRPC framework.

Other gRPC features provisioning for deadlocks, timeouts, cancellations, and error handling. For a complete list of gRPC features and supported functinanlity, we refer the reader to the book by Indrasiri and Kuruppu [64].

14.5 Others

Beyond the distributed programming frameworks detailed in the above sections, specific application domains benefited from application specific programming frameworks. Some such include Erlang [6], Emerald, Argus, and Orleans, that are based on message passing. There are also distributed programming languages based on the distributed shared memory model. Some of these include Mirage, Orca, and Linda. For a brief introduction to these variants, we refer the reader to [90].

There are other domain specific distributed programming frameworks such as Pregel [86] and GraphX [110] for graph processing, Graphlab [83] and TensorFlow [3] for machine learning algorithms, ane the like. These frameworks allow programmers to write distributed programs for specific domains to express their program as a sqeuence of high-level steps that are supported by these frameworks. This reuslts in higher productivity and also higher performance as the high-level steps are usually well-optimized.

14.6 Summary

There exist multiple languages and frameworks for programming disributed systems. This chapter did not discuss issues such as best practices in distributed programming, the idea of microservices, software architectures for distributed programming, fault location and debugging, and many other associated issues. For instance, writing efficient distributed programs requires good understanding of problem decomposition, studying the impact of interprocess communication vis-a-vis local computation, process synchronization, and the like. The book by Quinn [101] discusses these techniques in detail.

In addition, debugging distributed programs is often challenging since it may be difficult to replicate a scenario leading to a fault as events run across multiple processors.

14.7 Questions

1. Understand the difference bewteen HTTP/1.0 and HTTP/2.0 by preparing web pages that use the two protocols and the time taken to load webpages using these two protocols.

2. Study the benefits of client-side, server-side, and bidirectional streaming in gRPC using suitable examples. As part of this experiment, you can study the transfer time and the connection establishment time and hence understand the savings introduced by using the streaming model.

3. Discuss why a platform such as MPI does not provide any support for streaming connections like gRPC.

4. Think of problems that are very suitable to solve with each of MPI, Map-Reduce, and gRPC, and discuss why. Similarly, identify problems that are most unsuitable for the above frameworks and discuss why.

Chapter 15

Security in Distributed Systems

Chapter 16

Performance Evaluation and Benchmarking

Part IV

Case Studies

Bibliography

[1] Giraph, 2016. http://giraph.apache.org/.

[2] Bureau International des Podis et Mesures. https://www.bipm.org/en/, 2023.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, 2016.

[4] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: An experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[5] Apache Software Foundation. Hadoop.

[6] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

[7] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. *Sublinear Algorithms for (Δ + 1) Vertex Coloring*, pages 767–786. 2019.

[8] Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *30th Annual Symposium on Foundations of Computer Science*, pages 364–369. IEEE Computer Society, 1989.

[9] Baruch Awerbuch and David Peleg. Sparse partitions. In *31st Annual Symposium on Foundations of Computer Science*, pages 503–513. IEEE Computer Society, 1990.

[10] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., USA, 1986.

[11] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL system. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 331–343. ACM, 2017.

[12] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR*, pages 223–234, 2011.

[13] S. Bandyapadhyay, T. Inamdar, S. Pai, and S. V. Pemmaraju. Near-optimal clustering in the k-machine model. In *Proc. ICDCN*, 2018.

[14] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45, 2016.

[15] Surender Baswana and Sandeep Sen. A simple linear time algorithm for computing a (2k-1)-spanner of o($n^{1+1/k}$) size in weighted graphs. In *Automata, Languages and Programming, 30th International Colloquium, ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 384–296. Springer, 2003.

[16] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[17] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI Proceedings*, pages 47–60. USENIX Association, 2010.

[18] Andrew Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[19] Tushar Bisht, Kishore Kothapalli, and Sriram V. Pemmaraju. Superfast t-ruling sets. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 379–381. ACM, 2014.

[20] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. World Wide Web Consortium (W3C) Whitepaper, 2000.

[21] Fredereick Brooks. No silver bullet – essence and accident in software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[22] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350. USENIX Association, 2006.

[23] Cassandra. Apache cassandra. https://cassandra.apache.org/˙/index.html, 2010.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[25] Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The complexity of (δ+1) coloring in congested clique, massively parallel computation, and centralized local computation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, page 471–480, 2019.

[26] F. Chung and O. Simpson. Distributed algorithms for finding local clusters using heat kernel pagerank. In *Proc. WAW*, pages 77–189, 2015.

[27] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John K. Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 37–48. USENIX Association, 2013.

[28] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[29] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[32] Artur Czumaj, Jakub Łacki, Aleksander Madry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. page 471–484, 2018.

[33] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[35] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[36] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[37] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communication of the ACM*, 19(11):624–633, nov 1976.

[38] Facebook. https://engineering.fb.com/2020/03/18/production-engineering/ntp-service/, 2020.

[39] R. Fathi, A. R. Molla, and G. Pandurangan. Efficient distributed algorithms for the k-nearest neighbors problem. In *SPAA*, pages 527–529. ACM, 2020.

[40] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, 1988.

[41] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* Phd thesis, Univeristy of California, Irvine, 2000.

[42] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[43] Gnutella Developer Forum. The annotated gnutella protocol specification v0.4. `https://rfc-gnutella.sourceforge.net/developer/stable/`. Accessed on 26^{th} June 2024.

[44] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*, pages 134–141. IEEE Computer Society, 1990.

[45] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.

[46] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences).* W. H. Freeman, 1979.

[47] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. pages 129–138, 2018.

[48] Mohsen Ghaffari, Ce Jin, and Daan Nilis. A massively parallel algorithm for minimum weight vertex cover. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, page 259–268, 2020.

[49] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In *IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1650–1663, 2019.

[50] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019.

[51] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[52] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas E. Anderson. Scalable consistency in scatter. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 15–28. ACM, 2011.

[53] Google. grpc.

[54] Jim Gray. Notes on data base operating systems. In Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle, editors, *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer, 1978.

[55] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[56] The OMG Group. *Common Object Request Broker Architecture*. The Object Managment Group, 1991.

[57] X. F. Group. The xfs linux wiki, 2018.

[58] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[59] Nicholas J. A. Harvey, Christopher Liaw, and Paul Liu. Greedy and local ratio algorithms in the mapreduce model. In *Proceedings of*

the ACM Symposium on Parallelism in Algorithms and Architectures, SPAA, page 43–52, 2018.

[60] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[61] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, 2021.

[62] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system (extended abstract). In Les Belady, editor, *Proceedings of the Eleventh ACM Symposium on Operating System Principles, SOSP 1987, Stouffer Austin Hotel, Austin, Texas, USA, November 8-11, 1987*, pages 1–2. ACM, 1987.

[63] T. Inamdar, S. Pai, and S. V. Pemmaraju. Large-scale distributed algorithms for facility location with outliers. In *Proc. OPODIS*, 2018.

[64] K. Indrasiri and D. Kuruppu. *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[65] C. Jard and G.V. Jourdan. Dependency tracking and filtering in distributed computations. Technical Report 851, IRISA: Publications internes: Institut de Recherche en Informatique et Systèmes Aléatoires, 1994.

[66] Lujun Jia, Rajmohan Rajaraman, and Torsten Suel. An efficient distributed algorithm for constructing small dominating sets. *Distributed Comput.*, 15(4):193–205, 2002.

[67] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[68] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[69] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proc. ACM SPAA*, pages 938–948, 2010.

[70] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[71] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 391–410. SIAM, 2015.

[72] C. Konrad, S. V. Pemmaraju, T. Riaz, and P. Robinson. The complexity of symmetry breaking in massive graphs. In *Proc. DISC*, volume 146, pages 26:1–26:18, 2019.

[73] Kishore Kothapalli, Shreyas Pai, and Sriram V Pemmaraju. Sample-and-gather: Fast ruling set algorithms in the low-memory mpc model. In *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 182, pages 28–46, 2020.

[74] Kishore Kothapalli and Sriram V. Pemmaraju. Distributed graph coloring in a few rounds. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 31–40. ACM, 2011.

[75] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 300–309. ACM, 2004.

[76] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Proceedings*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2014.

[77] L Lamport. The part-time parliament. acm transactions on computer systems, 1998.

[78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[79] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[80] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. ACM, 2019.

[81] Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In Margo I. Seltzer and Richard Wheeler, editors, *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*, pages 153–166. USENIX, 2009.

[82] Barbara Liskov and Rivka Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. Association for Computing Machinery, 1986.

[83] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. UAI'10, page 340–349, 2010.

[84] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15(4):1036–1053, 1986.

[85] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[86] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 135–146. ACM, 2010.

[87] Udi Manber and Sun Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, page 4, USA, 1994. USENIX Association.

[88] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Paral lel and Distributed Algorithms*, pages 120–134, 1988.

[89] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[90] Heather Miller, Nat Dempkowski, James Larisch, and Christopher Meiklejohn. *Distributed Programming.* Available at https://github.com/heathermiller/dist-prog-book, 2017.

[91] David Mills. Network Time Protocol (NTP). RFC 958, 1985.

[92] Annamalai Muthu. Zippydb: A modern, distributed keyvalue data store. Retrieved from https://www.youtube.com/watch?v=DfiN7pG0D0k, 2015.

[93] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[94] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., 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 *19th USENIX Conference on File and Storage Technologies, FAST*, pages 217–231. USENIX Association, 2021.

[95] Alessandro Panconesi and Aravind Srinivasan. On the complexity of distributed network decomposition. *Journal of Algorithms*, 20(2):356–374, 1996.

[96] G. Pandurangan, P. Robinson, and M. Scquizzato. On the distributed complexity of large-scale graph computations. In *Proc. ACM SPAA*, pages 405–414, 2018.

[97] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and MST in large graphs. In Christian Scheideler and Seth Gilbert, editors, *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 429–438. ACM, 2016.

[98] Swapnil Patil, Garth A. Gibson, Samuel Lang, and Milo Polte. GIGA+: scalable directories for shared file systems. In Garth A. Gibson, editor, *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07), November 11, 2007, Reno, Nevada, USA*, pages 26–29. ACM Press, 2007.

[99] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, editors, *5th USENIX Conference on File and Storage Technologies, FAST 2007, February 13-16, 2007, San Jose, CA, USA*, pages 17–28. USENIX, 2007.

[100] J. Qiu, S. Jha, A. Luckow, and G. C. Fox. Towards HPC-ABDS: an initial high-performance big data stack. 2014. Available: http://grids.ucs.indiana.edu/ptliupages/publications/nist-hpc-abds.pdf.

[101] M. J. Quinn. *Parallel Programming in C with Mpi and Openmp*. McGraw Hill, 2017.

[102] Ronald L. Rivest. The MD5 message-digest algorithm. *RFC*, 1321:1–21, 1992.

[103] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12–16, 2001 Proceedings 2*, pages 329–350. Springer, 2001.

[104] Russel Sandberg. The sun network filesystem: Design, implementation and experience. In *USENIX Technical Conference and Exhibition*, pages 1–16, 1986.

[105] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In Darrell D. E. Long, editor, *Proceedings of the FAST '02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA*, pages 231–244. USENIX, 2002.

[106] SimpleDB. Amazon simpledb. https://aws.amazon.com/simpledb/, 2007.

[107] Mukesh Singhal and Ajay D. Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 43(1):47–52, 1992.

[108] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 729–730. ACM, 2012.

[109] Dale Skeen. A quorum-based commit protocol. Technical Report TR-82-483, Cornell University, 1982.

[110] Apache Spark. Apache graphx. https://spark.apache.org/graphx/, 2014.

[111] Stern and Labiagga. *Managing NFS and NIS*. Orielly, 2 edition, 2001.

[112] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[113] Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 1999.

[114] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, page 1–12, 2012.

[115] VoltDB. Voltdb resources. http://voltdb.com/resources/whitepapers, 2012.

[116] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2 edition, 2000.

[117] Lustre Wiki. https://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf, 2017.

[118] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. pages 233–242. Association for Computing Machinery, 1984.

[119] Grigory Yaroslavtsev and Adithya Vadapalli. Massively parallel algorithms and hardness for single-linkage clustering under ℓ_p distances. In *Proceedings of the 35th International Conference on Machine Learning*, pages 5600–5609, 2018.

Index