Introduction to Consensus in Distributed Systems

Explore what we'll study in the upcoming chapters regarding consensus in distributed systems.

We'll cover the following

Motivation

Consensus is one of the oldest problems in designing large-scale systems, and we have many algorithms to achieve consensus under different circumstances. We need consensus in our systems for many tasks, such as electing a leader in primary-secondary configurations, consistent data replication across many replicas of a database or filesystem, distributed locking, distributed transactions, atomic broadcast, and more. It will not be an exaggeration to say that every large-scale distributed system uses consensus frequently in daily operations. In this section, we'll study the theoretical foundations and practical algorithms to achieve consensus when nodes or networks can fail in different ways.

Consensus algorithms

We will discuss three practical consensus algorithms—Two-phase commit, Paxos, and Raft. These algorithms are commonly used in many systems, such as databases and replicated data stores. These consensus algorithms borrow heavily from decades of research to achieve consensus in a fault-tolerant manner. To better understand the practical consensus algorithms, we have included two chapters on consensus fundamentals and how to achieve safe replication using state machines. Let's discuss high-level details of our content.

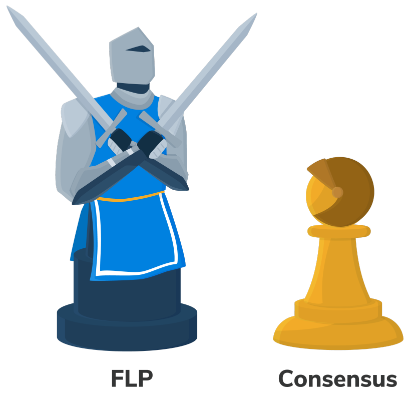
Consensus fundamentals (Two Generals' Problem, the FLP impossibility, Byzantine Generals Problem)

Our first chapter establishes the baseline for distributed consensus on what is possible and what's not under different computational models. This chapter also formally defines the necessary terminology and provides a lens through which we can understand the chapters on two-phase commit, Paxos, and Raft.

The Two Generals' Problem tells us that consensus is impossible in a synchronous model if the network can lose messages. The fundamental difficulty is that the sender of a message can't tell if the recipient received the message or not. This result seems unintuitive and shocking. This is why consensus is not possible even in a friendly computational environment of a synchronous model. We'll discuss the rationale behind this impossibility and explore what can be done.

The FLP result tells us that in an asynchronous environment, no deterministic algorithm exists to achieve consensus between processes even if a single process fails (crashes). By consensus, we mean we want all the processes to agree on a common value. For someone unfamiliar with the FLP result, a few questions arise. For example, since the implications of the FLP result may appear unfavorable, what advantages or benefits does it provide? FLP says the consensus is impossible, but how do major services consistently replicate data across different data centers?

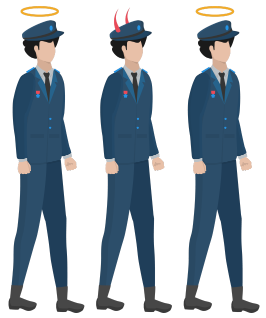
The first question is how FLP is good for us. As an analogy, FLP can be compared to NP-complete problems in algorithms. If someone hands us a task to devise a fault-tolerant consensus algorithm in an asynchronous environment, we can point to FLP and say this task is not doable. However, just like many real-world problems are NP-complete, there are many problems where we need a consensus algorithm. Our second point will address how to get free from the limitations set by FLP.



FLP stands on certain assumptions, one of which is the asynchronous environment. One sure-shot way to undermine any theorem is to attack its assumptions. In the case of FLP, if we could run our processes in a synchronous environment, we could achieve consensus. The global Internet is neither purely asynchronous (fortunately!) nor purely synchronous—it keeps moving around these two states over time. This allows us to devise a consensus algorithm that is always correct (using safety conditions) but might not make progress when we are in a bad asynchronous state (liveness conditions).

Additionally, for some problems, a simple majority of processes is enough instead of *everyone* agreeing on something immediately at the moment. Most data replication algorithms work on such a principle where they achieve a majority quorum to do their work, and any nodes left behind eventually catch up. Doing so helps us progress even under a few faults. For example, we can tolerate two node failures in a consensus cluster of five nodes because the remaining three will be a majority.

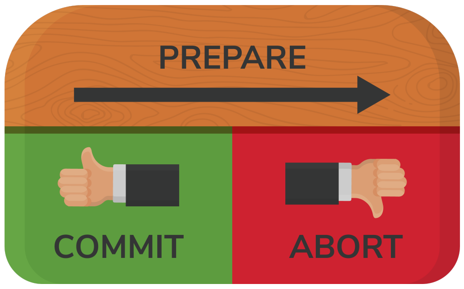
The Byzantine Generals' Problem tells us that if a network is non-faulty and the computational model is at least partially synchronous, we can tolerate Byzantine faults by keeping a specific number of nodes in the system. The challenge here will be to



ensure there are at least two-thirds of non-faulty nodes in a replica group. Doing so can be challenging because detecting Byzantine behavior is not always possible.

Two-phase commit

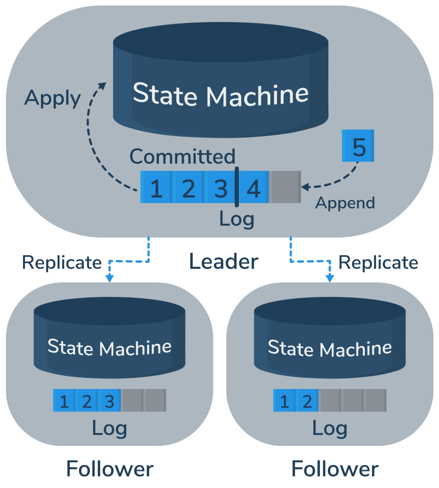
The two-phase commit (2PC) protocol is a widely used consensus protocol for achieving atomic commit in a distributed system to make sure that all nodes either commit or abort entirely. 2PC works in two steps. In the first step, the agreed-upon value is shared and votes are gathered. In the second step, nodes simply turn on a switch to show the outcome of the first step.



For 2PC, the consensus means that everyone is on the same page. We usually don't rely on a simple majority because 2PC is often used in scenarios where we want to do a different thing at different places (for example, mutations at different shards of a database at different sites for a transaction). Under asynchronous conditions, 2PC can stop making progress, but it is always safe. (Due to FLP, we know why this is the case.) We will see that a variant of 2PC (three-phase commit or 3PC) tries to improve on the shortcomings of 2PC but violates safety conditions when the network is asynchronous. We should use great caution while using algorithms that can violate safety under certain conditions or carefully understand the assumptions of a system before using it.

State machine replication

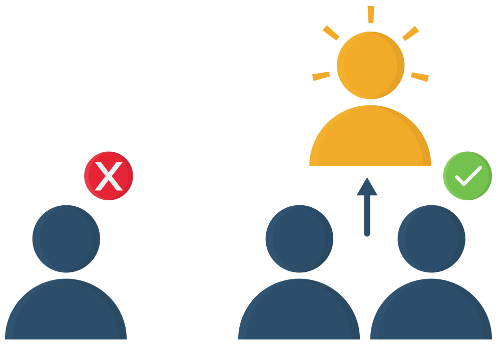
Our goal is to make fault-tolerant systems. A large variety of computations can be modeled as a state machine, which starts at some valid initial state. Each new command moves the machine to the next valid state, and so on. Replicated databases at different sites are an example of a state machine, where each replica starts at a good state. Commands (such as update, write, delete, etc.) are communicated to all replicas in the same order, and independent state machines apply those commands on the data.



The SMR paper shows us the necessary conditions to build fault-tolerant state machines that are at the heart of modern consensus algorithms such as Paxos and Raft.

Paxos

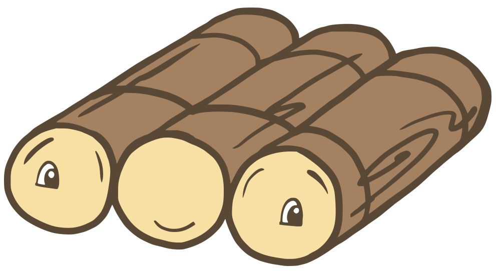
Paxos is a widely used consensus algorithm that uses a simple majority to make progress. Paxos is always safe but not live when a majority of nodes are not available. Leslie Lamport presented Paxos in his landmark paper The Part-Time Parliament. Lamport later attempted to present the idea in a simpler way through his paper “Paxos Made Simple”.



Paxos algorithm has a reputation for being difficult to understand primarily because the core of the paper focuses mostly on the correctness proof of the algorithm without providing any engineering guidance for its implementation. Real projects have encountered difficulties when implementing Paxos, and they have found many practical aspects that were under-specified in the original algorithm. Because of that, many engineering teams were forced to devise their own Paxos extensions. (The paper "Paxos Made Live - An Engineering Perspective" is one such effort where Google implemented a variant of Paxos for their Chubby service.) We will present a simple variant of the algorithm in our Paxos chapter.

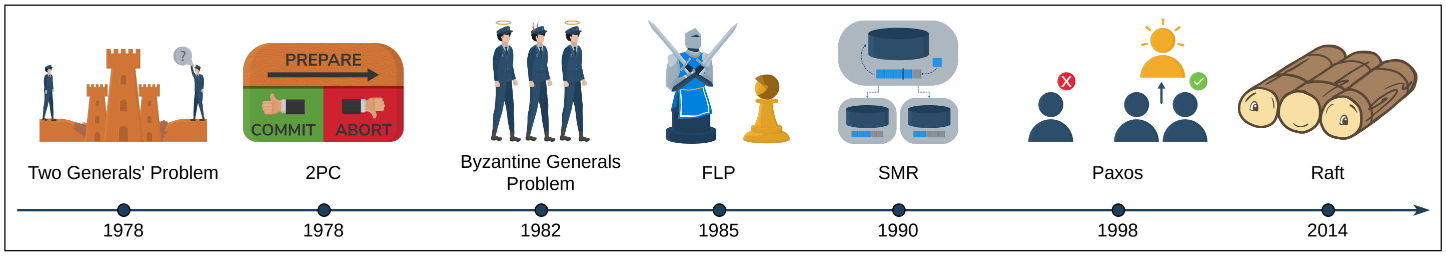
Raft

Due to continued dissatisfaction among engineers with the complexity of the Paxos algorithm, Diego Ongaro and John Ousterhout invented the Raft algorithm. The authors believe that the understandability of a consensus algorithm is key if we want to enable engineers to extend the algorithm for their use cases. Additionally, Raft provides practical guidance on the implementation of the algorithm. Raft has been getting traction recently. Like Paxos, the Raft algorithm needs a simple majority to work.



Raft is always safe, but progress can halt if a majority is unavailable. Raft makes a few optimizations (such as having a leader as an integral part of the algorithm) to improve the algorithm regarding the number of messages communicated between the nodes and the resulting reduced time to reach consensus. Many of Raft's design decisions were influenced by ease of understanding for engineers. After learning Paxos and Raft, we will be better positioned to decide which is easier to understand.

We hope our selection of consensus systems will provide you with many important lessons in system design. Let's dive in!



Evolution of consensus algorithms and their related concepts

Hide Original Sources of Consensus Algorithms

We have distilled information from many sources and presented it in an easy-to-learn manner in this section. Following are some of the sources that have influenced our presentation.

[Two Generals' Problem] Michael J. Flynn, Jim N. Gray, A. K. Jones, K. Lagally H. Opderbeck, Gerald J. Popek, B. Randell, J. H. Saltzer, H. R. Wehle. Lecture Notes in Computer Science, Operating Systems, An Advanced Course, Springer-Verlang, Berling Heidelberg New York, 1978.

[FLP] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2 (April 1985), 374–382.

[Byzantine Generals Problem] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems 4, no. 3 (1982): 382-401.

[State Machine Replication (SMR)] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22, 4 (Dec. 1990), 299–319.

[Two-phase Commit (2PC)] Jim Gray. Notes on Data Base Operating Systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, Operating Systems: An Advanced Course, volume 60 of Lecture Notes in Computer Science, pages 393–481. Springer-Verlag, Berlin, Heidelberg, New York, 1978.

[Paxos] Leslie Lamport. Paxos Made Simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) (2001): 51-58.

[Paxos-Live] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing, pp. 398-407. 2007.

[Raft] Diego Ongaro, and John Ousterhout. In Search of an Understandable Consensus Algorithm. In 2014 USENIX Annual Technical Conference, pp. 305-319. 2014.

Back

Quiz on Kafka

Next

Consensus Prerequisit...