

2022-02-16

Title to be decided

David Kleingeld

Email

Contents

1 Background	2
1.1 Distributed Computing	2
1.2 Faults and Delays	2
2 Consensus Algorithms	3
3 File System	7
4 Existing distributed file systems	7

1 Background

1.1 Distributed Computing

When state of the art hardware is no longer fast enough to run a system the only option is scaling out. Then there is a choice, do you buy expensive, reliable high performance supercomputer or commodity servers connected by Ip and ethernet? This is the choice between High Performance (HPC) and Distributed Computing. With HPC faults in the hardware are rare and can be handled by restarting, easing development. In a distributed context faults are the norm, restarting the entire system is not an option or you would be down all the time. Resilience against faults comes at an, often significant, cost to performance. It may also limit scalability. As the scale of a system increases so does the frequency with which one of the parts fails. Even the most robust part will fail and given enough of them the system will fail frequently. Therefore at very large scales HPC is not even an option.

1.2 Faults and Delays

Before we can build a fault resistant system we need to know what we need to keep in mind. While hardware failures are, the norm in distributed computing, faults are not the only issue to keep in mind.

It is entirely normal for the clock of a computer to run slightly to fast or to slow. The drift will be tens of milliseconds [1] unless special measures are taken¹. Worse a process can be paused and then resumed at any time. Such a pause could be because the process thread is pre-empted, because its virtual machine is paused or because the process was stopped and resumed after a while².

In a distributed system the computers that form the system or *the nodes*, are connected by IP over ethernet. Ethernet gives no guarantee a packet is delivered on time or at all. A node can be unreachable before suddenly working fine again.

¹One could synchronize the time within a datacenter or provide nodes with more accurate clocks

²On linux by sending SigStop then SigCont

A system model is an abstraction defining what an algorithm can assume. Regarding timing there are three models.

1. The Synchronous model allows an algorithm to assume that clocks are synchronized within some bound and network traffic will arrive within a fixed time.
2. The Partially synchronous model is a more realistic model. Most of the time clocks will be correct within a bound and network traffic will arrive within a fixed bound. However sometimes clocks will drift unbounded and some traffic might be delayed forever.
3. The Asynchronous model has no clock, it is very restrictive.

For most distributed systems we work with the Partially Synchronous model. We assume hardware faults cause a crash from which the node can be recovered later. Either automatically as it restarts or after maintenance.

2 Consensus Algorithms

In this world where the network can not be trusted, time lies to us and servers will randomly crash and burn how can we get anything done at all? Let's discuss how we can build a system we can trust, a system that behaves *consistently*. To build such a system we need the parts that make up the system to agree with each other, they must have *Consensus*. Here I discuss three well known solutions. Before we get to that let's look at the principle that underlies them all: *The truth is defined by the majority*.

Quorums

Imagine a node hard at work processing requests from its siblings, suddenly it gets pre-empted. The other nodes notice it is no longer responding and declare it dead, they don't know its threads got paused. A few seconds later the node responds again as if nothing had happened, and in truth, unless it checks the system clock, from its perspective no time has passed. Alternatively a network error might partition the system, each group of servers can reach each other but not the others. The nodes in the group will declare those in the other group dead and continue their work. Usually this results in data loss if the work progresses at all.

We can solve this by voting over each decision. It will be a strange vote, no node cares about the decision itself. In most implementations the nodes only check if it regards the sender as trustworthy or alive and then vote yes. To prove liveness the vote proposal could include a number. Voters only vote yes if the number is correct. For example if the number is the highest they have seen. If a majority votes yes the node that requested the vote can be sure it's not dead or disconnected. This is the idea behind "Quorums," majorities of nodes that vote.

Paxos

The Paxos algorithm[2] uses a quorum to provide consensus. It does this by choosing a value among proposals such that only that value can be read as the accepted value. Usually it is used to build a fault tolerant distributed state machine.

In Paxos there are three roles: proposer, acceptor and learner. It is possible for nodes to fulfill only one or two of these roles. For the rest of this explanation assume each node fulfills all three. To reach consensus on a new value we go through two phases: prepare and accept. Once the majority of the nodes has accepted a proposal the value of that proposal has been chosen. In Paxos nodes keep track of the highest proposal number n they have seen. Let's go through a Paxos iteration from the perspective of a node trying to share something, *a value*.

In the first phase a new *value* is proposed by our node. It sends a *prepare* request to a majority of acceptors. The request contains a proposal number n higher than the highest number our node has seen up till now. The number is unique to our node. Each acceptor only responds if our number n is the highest it has seen. If an acceptor had already accepted one or more requests it includes the accepted proposal with the highest n in its response.

In phase two our node checks if it got a response from the majority. Our node is going to send an accept request back to those nodes. The content of the accept request depends on what our node received in response to its prepare request:

1. Our node received a response with number n_p . This means an acceptor has already accepted a value. If we continued with our own value the system would have two different accepted values. Therefore the content of our accept request will be the value from proposal n_p .

2. It received only acknowledging replies and none contained a previously accepted value. The system has not yet decided on a value. The content of our accept request will be the value or node wants to propose but with our number n .

The acceptors accept the request if they did not yet receive a prepare request numbered greater than n . On accepting a request an acceptor sends a message to all learners³. This way the learners learn a new value as soon as its ready.

Let's get a feeling why this works by looking at what happens during node failure. Imagine a case where a minimal majority m accept value v_a . A single node in m freezes after the first learners learned of the now chosen value v_a . After freezing $m - 1$ of the nodes will reply v_a as value to learners. The learners will conclude no value has been chosen given $m - 1$ is not a majority⁴. As seen above acceptors change their value if they receive a higher numbered accept request. If a single node changes its value to v_b consensus will break since v_a has already been seen as the chosen value by a learner. A new proposal that can result into higher numbered accept requests needs a majority response. A majority response will include a node from $m - 1$. That node will include v_a as the accepted value. The value for the accept request changes to v_a . No accept request with another value than v_a can thus be issued. Another value v_b will therefore never be accepted. The new accept by issued to a majority changing at least one node to have accepted v_a . Now at least $m + 1$ nodes have v_a as accepted value.

To build a distributed state machine you run multiple instances of Paxos. This is often referred to as MULTI-PAXOS. The value for each instance is a command to change the shared state. Unfortunately multi paxos is not specified in literature and has never been verified.

Raft

The Paxos algorithm allows us to reach consensus on a single value. The Raft algorithm allows us to share a log between nodes. We can only append to and reading from the. That is the log will always be the same on all nodes. As long as a majority of the nodes still function the log will be readable and appendable.

³remember usually every node is a learner

⁴this is not yet inconsistent, Paxos does not guarantee consistency over whether a value has been chosen

In Raft there is always a single leader. The leader is determined using a *quorum*. Appending to the log is sequential because only the leader is allowed to append to the log. There are two parts to Raft, *electing leaders* and *log replication*.

Leader election A Raft[3] cluster starts without a leader and when one is elected it can fail at any time. Nodes in raft start as followers, monitor the leader by waiting for heartbeats. If a follower does not receive a heartbeat on time it will try to become the leader, it becomes a candidate. In a fresh cluster without a leader one or more nodes will switch to candidates.

A candidate tries to get itself elected. For that it needs the votes of a majority of the cluster. It first asks all servers for their vote. If a majority responds with their vote the candidate becomes a leader. When there are multiple candidates the votes can split, no candidate reaches a majority. When it takes too long to receive a majority of the votes the candidate starts a fresh election. A candidate loses the election as soon as it receives a heartbeat from a *valid* leader.

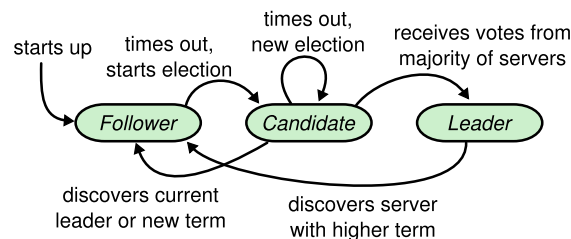


Figure 1: Node states. Most of the time all nodes except one are followers. One node is a leader. As failures are detected by time outs the nodes change state. Adjusted from [3]

Log replication

Consensus as a service

Test text

3 File System

A file system is split into two parts, the files and the directory structure. File properties, or metadata, such as its name, identifier, size etc are stored in the directory. Typically the directory entry itself only contains the file name and its unique identifier. Using the identifier the other metadata for the file can be fetched. The content of the file is split into blocks these blocks are stored on stable storage such as an hard drive or ssd. The file system defines an API to allow modifying the files system providing ways to *create*, *read*, *write*, *seek* and *truncate* files.

Usually the system adds a distinction between open and closed files. The apis *read* *write* and *seek* are then only allowed on open files. This makes it possible to provide some concistency guarentees in a concurrent envirement. For example allowing a file to be opened only if it was not already open. This can prevent a user from corrupting data by writing from multiple processes at the same place in the file. There is no risk to reading the same file from multiple process, even while appending to it from other processes⁵. To allow such use a file systems can define opening a file in read-only, append-only or read-write mode. On Linux this is opt in⁶. Even more semantics exist for example allowing opening multiple non overlapping ranges of a file for writing.

4 Existing distributed file systems

Network File System

Often we want to share filesystems over a network to share files using a *Network file system*. These integrate in the interface of the client. A widely supported system is NFS. In NFS a part of a local directory is exported/shared by a local NFS-server. Other machines can then connect and overlay part of their directory with the exported one. The NFS protocol forwards file operations from the client to the host over the network. When an operation has been applied on the host the result is traced back to the client. To increase performance the client (almost always) caches file blocks and metadata.

In a shared envirement it is commanplace for multiple users to simultaneously

⁵The OS can ensure append writes are serialized, this is usefull for writing to al log file where each write call appends an entire log line to a file opened in append mode

⁶see flock or fcntl or mandatory locking

access the same files. In NFS this can be problematic, as meta data is cached new files can appear to other users after 30 seconds. Further more simultaneous writes can become interleaved as each write gets split into multiple network packets [5, p. 527], writing corrupt data. Version 4 improves the semantics respecting unix advisory file locks [4]. Most applications do not take advisory locks into account still risking data corruption.

Google file system

Hadoop FS

Ceph, subtree partitioning

References

- [1] M Caporali and R Ambrosini. “How closely can a personal computer clock track the UTC timescale via the internet?” In: *European Journal of Physics* 23.4 (June 2002), pp. L17–L21. doi: 10.1088/0143-0807/23/4/103. URL: <https://doi.org/10.1088/0143-0807/23/4/103>.
- [2] Leslie Lamport. “Paxos Made Simple”. In: *ACM SIGACT News (Distributed Computing Column)* 32, 4 (Whole Number 121, December 2001) (Dec. 2001), pp. 51–58. URL: <https://www.microsoft.com/en-us/research/publication/paxos-made-simple/>.
- [3] Diego Ongaro and Ousterhout John. *In Search of an Understandable Consensus Algorithm (Extended Version)*. <https://raft.github.io/>. accessed 15-Feb-2022. 2014.
- [4] S Shepler et al. *Network File System (NFS) version 4 Protocol*. RFC 3530. IETF, Apr. 2003. URL: <https://www.ietf.org/rfc/rfc3530.txt>.
- [5] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne. *Operating system concepts*. John Wiley & Sons, 2014.