2022-03-02

Title to be decided

David Kleingeld

Email

Contents

1 Background

TODO write tiney intro

1.1 Distributed Computing

When state of the art hardware is no longer fast enough to run a system the option that remains is scaling out. Here then there is a choice, do you use an expansive, 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, simplifying software. In a distributed context faults are the norm, restarting the entire system is not an option or you would be down all the time. Resiliance against faults comes at an, often significant, cost to performance. Fault tolerance may limit scalability. As the scale of a system increases so does the frequancy with which one of the parts fails. Even the most robust part will fail and given enough of them the system will fail frequantly. 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 can rely on. 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 resulting drift will normally be tens of milliseconds [1] unless special measures are taken[1]. Event worse a process can be paused and then resumed at any time. Such a pause could be because the process thread is pre-emted, because its virtual machine is paused or because the process was paused and resumed after a while[2].

In a distributed system the computers (*nodes*)that form the system are connected by IP over ethernet. Ethernet gives no guarentee a packet is

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

[2] On linux by sending SigStop then SigCont

deliverd on time or at all. A node can be unreachable before seemingly working fine again.

Using a system model we formalize the faults that can occur. For timing there are three models.

1. The Synchronous model allowes an algorithm to assumes 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 assume the Partially Synchronous model. Hardware faults cause a crash from which the node can be recoverd later. Either automatically as it restarts or after maintenance.

1.3 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? Lets discuss how we can build a system we can trust, a system that behaves *consistantly*. To build such a system we need the parts that make up the system to agree with eachother, the must have *Consensus*. Here I discuss three well known solutions. Before we get to that lets 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 stops responding. The other nodes notice it is no longer responding and declear it dead, they do not know its threads got paused. A few seconds later the node responds again as if nothing had happend, and unless it checks the system clock, no time has paused from its perspective. Or imagine a network fault partitions the system, each group of servers can reach its members but not others. The nodes in the group will declear those in the other group dead and continue their work. Both these scenarios usually result in data loss, if the work progresses at all.

We can prevent this by voting over each descision. It will be a strange vote, no node cares about the descision itself. In most implementations a node only checks if it regards the sender as trustwothy or alive and then vote yes. To prove liveliness 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 is, at that instance, not dead or disconnected. This is the idea behind "Quorums," majorities of nodes that vote.

Paxos

The PAXOS algorithm [5] uses a quorum to provide concensus. It enables us to chosing a single 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 fullfil only one or two of these roles. Usually, and for the the rest of this explanation each node fullfils 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 included in that proposal has been chosen. Nodes keep track of the highest proposal number n they have seen.

Lets 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 *perpare* request to a majority of acceptors. The request contains a proposal number n higher then the highest number our node has seen up till now. The number is unique to our node[3]. 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 respons.

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 recieved in response to its prepare request:

1. It recieved a response with number n_p. This means an acceptor has already accepted a value. If we continued with our own value the system

[3] This could be nodes incrementing the number by the cluster size having initally assigned numbers 0 to *cluster_size*

would have two different accepted values. Therefor the content of our accept request will be the value from proposal n_p.

2. It recieved only acknowleding 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.

Each acceptor accepts the request if it did not yet recieve a prepare request numberd greater then n. On accepting a request an acceptor sends a message to all learners[4]. This way the learners learn a new value as soon as its ready.

Lets 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 fails by pausing 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 majority[5]. Acceptors change their value if they recieve a higher numberd 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 numberd accept requests needs a majority response. A majority response will include a node from $m-1$. That node will incude v_a as the accepted value. The value for the accept request then changes to v_a. No accept request with another value then v_a can thus be issued. Another value v_b will therefore never be accepted. The new accept request is issued to a majority adding at least one node to those having 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 referd to as MULTI-PAXOS. The value for each instance is a command to change the shared state. 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 enables consent on a shared log. We can only append to and

[4] remember every node is a learner

[5] this is not yet inconsistent, PAXOS does not guarentee consistency over wether a value has been chosen

reading from the log. The content of the log wil always be the same on all nodes. As long as a majority of the nodes still function the log will be readable and appendable.

Raft maintains a single leader. Appanding to the log is sequential because only the leader is allowed to append. The leader is decided on by a *quorum*. There are two parts to raft, *electing leaders* and *log replication*.

Leader election A raft [7] cluster starts without a leader and when it has a leader it can fail at any time. The cluster therefore must be able to reliable decide on a new leader at any time. Nodes in raft start as followers, monitoring the leader by waiting for heartbeats. If a follower does not recieve a heartbeat from a *valid* leader on time it will try to become the leader, it becomes a candidate. In a fresh cluster without a leader one or more nodes become candidates.

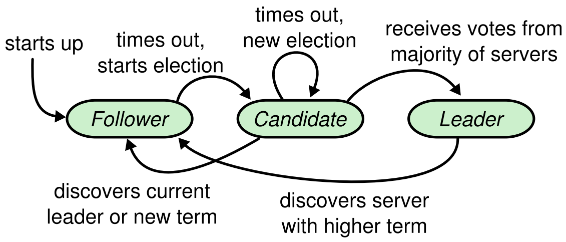

Figure 1: *A raft node states. Most of the time all nodes exept one are followers. One node is a leader. As failures are detected by time outs the nodes change state. Ajusted from [7].*

A candidate tries to get itself elected. For that it needs the votes of a majority of the cluster. It asks all nodes for their vote. Note that servers vote only once and only if the candidate would become a *valid* leader. If a majority of the cluster responds to a candidate with their vote that candidate becomes the leader. If it takes to long to recieve a majority of the votes a candidate starts a fresh election. When there are multiple candidates requesting votes the vote might split[6], no candidate then reaches a majority. A candidate immidiatly loses the election if it recieves a heartbeat from a *valid* leader. These state changes are illustrated in fig. 1.

In raft time can be devided in terms. A term is a failed election where no node won or the period from the election of a leader to its failure, illustrated

[6] Election timeouts are randomised so this does not repeat infinitly.

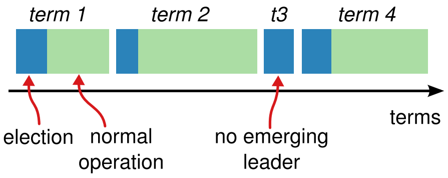

Figure 2: *An example of how time is devided in terms in a* RAFT *cluster. Taken from from [7].*

in fig. 2. Terms are used to determine the current leader, the leader with the highest terms. A heartbeat is *valid* if, as far as the reciever knows, it originates from the current leader. A message can only be from the *current* leader if the message *term* is equal or higher then the recieving nodes term. If a node recieves a message with a higher term it updates its own to be that of the message.

When a node starts its *term* is zero. If a node becomes a candidate it increments its *term* by one. Now imagine a candidates with a *term* equal or higher then that of the majority of the cluster. When recieving a vote request the majority will determine this candidate could become a *valid* leader. This candidate will get the majority vote in the absence of another candidate and become *the* leader.

Log replication To append an entry to the log a leader sends a append request to all nodes. Messages from invalid leaders are rejected. The leader knows an entry is committed after a majority of nodes acknowledged the append. For exampl entry 5 in fig. 3 is committed. The leader includes the index up to which entries are comitted in all its messages. This means entries will become comitted on all followers at the latest with the next heartbeat. If the leader approaches the timeout and no entry needs to be added it sends an empty append. There is no need for a special heartbeat message.

There are a few edge cases that require followers to be carefull when appending. A follower may have an incomplete log if it did not recieve a previous append, it may have messages the leader does not and finally we must prevent a candidate missing committed entries from becoming the leader.

1. To detect missing log entries, the entries are indexed incrementally. The leader includes the index of the previous entry and the term when it was

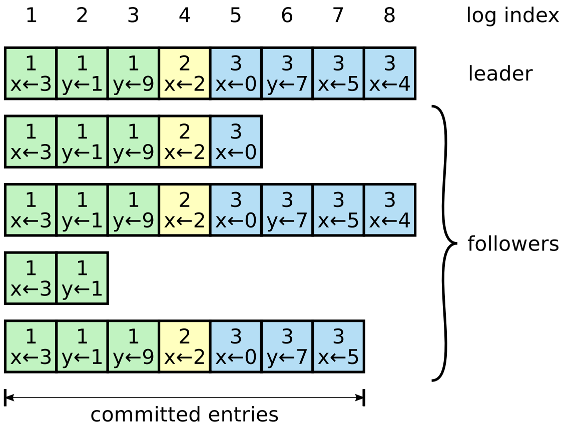
1 2 3 4 5 6 7 8
log index
1 x←3 1 y←1 1 y←9 2 x←2 3 x←0 3 y←7 3 x←5 3 x←4
leader
1 x←3 1 y←1 1 y←9 2 x←2 3 x←0
1 x←3 1 y←1 1 y←9 2 x←2 3 x←0 3 y←7 3 x←5 3 x←4
1 x←3 1 y←1
1 x←3 1 y←1 1 y←9 2 x←2 3 x←0 3 y←7 3 x←5
followers
committed entries

Figure 3: *Logs for multiple nodes. Each row is a different node. The log entries are commands to change shared variables x and y to different values. The shade boxes and the number at their top indictate the term. Taken from [7].*

appended in append requests. If a followers last log entry does not match the included index and term the follower responds with an error. The leader will send its complete log for the follower to duplicate[7].

2. When a follower has entries the leader misses these will occupy indices the leader wil use in the future. This happens when a previous leader crashed having pushed logs to some but not majority of followers. When the leader pushes a new entry with index k the follower wil notice it already has an entry with index k. At that point it simply overwrites what it had at k.

3. A new leader missing commited entries will push a wrong log to followers missing entries. For a entry to be committed it must be stored on the majority of the cluster. To win the election a node has to have the votes from the majority. Thus restricting followers to only vote for candidates that are as up-to-date as they are is enough. Followers thus do not vote for a candidate with a lower index then they themself have.

Log compaction Keeping the entire log is rather inefficient. Especially as nodes are added and need to get send the entire log. Usually raft is used to

[7] This is rather inefficient, in the next paragraph we will come back to this

build a state machine, in which case sending over the state is faster then the log. The state machine send is a snapshot of the system, only valid for the moment in time it was taken. All nodes take snapshots independently of the leader.

To take a snapshot a nodes writes the state machine to file together with the current *term* and *index*. It then discards all comitted entries up to the snapshotted *index* from its log.

Followers that lag far behind are now send the snapshot and all logs that follow. The follower wipes its log before accepting the snapshot.

Consensus as a service

So the problem of consensus has been solved, but the solutions are non trivial to implement. We need to shape our application to fit the models the solutions use. If we are using RAFT that means our systems must be build around a log. Then we need to build the application being carefull we implement the consensus algorithm correctly. A popular alternative is to use a coordination service. Here we look at ZooKeeper [3] an example of a wait free coordination service. I first focus on the implementation, drawing parallels to RAFT before we look at the *Api* ZooKeeper exposes.

A ZooKeeper cluster has a designated leader that replicates a database on all nodes. Only the leader can modify the database, therefore only the leader handles *write* requests. The nodes handle *read* requests themself, *write* requests are forwarded to the leader. To handle a *write* requests ZooKeeper relies on a consensus algorithm called Zab [4]. It is an atomic broadcast capable of crash recovery. It uses a strong leader quite similar to RAFT and guarentees that changes broadcast by the leader are delivered in the order they where send and after changes from previous (crashed) leaders. Zab can deliver messages twice during recovery. To account for this ZooKeeper turns change requests into *idempotent* transactions. Theses can be applied multiple times with the same result.

ZooKeeper exposes its strongly consistent database as a hierarchical name space (a tree) of *znodes*. Each znode contains a small amount of data and is identified by its *path*. Using the *Api* clients can operate on the znodes. They can:

1. Create new znodes
2. Change the data in a znode
3. Delete existing znodes
4. Sync with the leader
5. Query if a znode exists
6. Read the data in a znode
7. Get list of the children of the znode

The last three operations support watching, the client then gets a single notification when the result of the operation changed. The notification does not contain any information regarding the change and the value is no longer watched after the notification. Clients can use this to ensure locally cached data is up-to-date.

Clients communicating outside of ZooKeeper need to take special measures to ensure consistancy. For example let client A updates some znode from value v_1 to value v_2 then communicates to client B, through another medium then ZooKeeper. In this example client A and B are connected to different ZooKeeper nodes. If the communication from A causes B to read the znode it will get the previous value v_1 from ZooKeeper if its ZooKeeper node is lagging behind. To avoid this race condition client B can call *sync* first which will make the zookeeper node processes all outstanding requests from the leader before returning.

Raft has this same race condition. Intuitively it seems as we can just ensure a heartbeat has passed, by then all (functioning) node will be updated. However this is not enough, as a faulty node could be suspended and not notice it is outdated at all. Instead the solution is to include the last log index client A saw in its communication to client B. Paxos does not *need* to suffer from this problem but it can. In Paxos reading means asks a *learner* for the value, the *learner* can then ask the majority of the system if they have accepted a value and if so what it is. Usually paxos is optimized up by making acceptors inform learners of a change. In this case a leader that missed a message from an acceptor that there is a value will incorrectly return to client B there is no value.

1.4 File System

NOT YET REWRITTEN 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 *trunctate* 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 concistancy guarentees in a concurrent envirement. For example allowing a file to be opend 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 processes[8]. 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 in[9]. Even more semantics exist for example allowing opening multiple non overlapping ranges of a file for writing.

1.5 Distributed file systems

Here I will discuss the two most widely used destributed fily systems. We will look at how they work and the implementation. Before I get to that I will use a very basic file sharing system, Network File System, to illustrate why these distributed systems need their complexity.

Network File System

A basic way to share files is to expose a filesystems via a network to share. For this you use 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 simultaniously access the same files. In NFS this can be problamatic, as meta data is cached

[8] The 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 opend in append mode

[9] see flock or fcntl or mandatory locking

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 [9, p. 527], writing corrupt data. Version 4 improves the semantics respecting unix advisory file locks [8]. Most applications do not take advisory locks into account still risking data corruption.

Google file system

The Google File System [2] was developed in 2003 in a response to Googles rapidly growing search index which generated unusually large files [6]. The key to the system is the seperation of the control plane from the data plane. That is, the file data is stored on many *chunk servers* while a single server coordinates access to, location of and replicated of the data. The coordinating server also serves the metadata. Because all descisions are made on a single machine GFS needs no consensus algorithm. A chunk server simply executes all client requests as the coordinating server will already have decided if the request is allowed.

When a GFS client wants to operate on a file it contacts the main node for metadata. The client then uses the metadata to determine on which chunk servers the file content is located. Finally it streams bytes directly to or from the chunk servers. If multiple clients whish to operate on the same file one of the chunk servers serializes those requests to a fixed order. The resulting architecture can be seen in fig. 4.

To ensure data integrity data is checksummed and replicated over multiple servers. The replicas are carefully spread around to cluster to prevent a network switch or power supply failure taking all replicas offline and to ensure equal utilization of network and disks. The coordinating server re-creates lost chuncks as needed. The cluster periodically rebalances chunks between machines filling up newly added servers.

To speed up performance GFS provides a relaxed consistency model for the file data.

A single machine can efficiently handle all file metadata requests, as long as files are sufficiently large. If the cluster grows sufficiently large while the files stay small the metadata will no longer fit in the coordinating servers memory. Effectively GFS has a limit on the number of files. This limit became a problem as it was used for services with smaller files. To work around this applications packed smaller files together before submitting the bundle as a

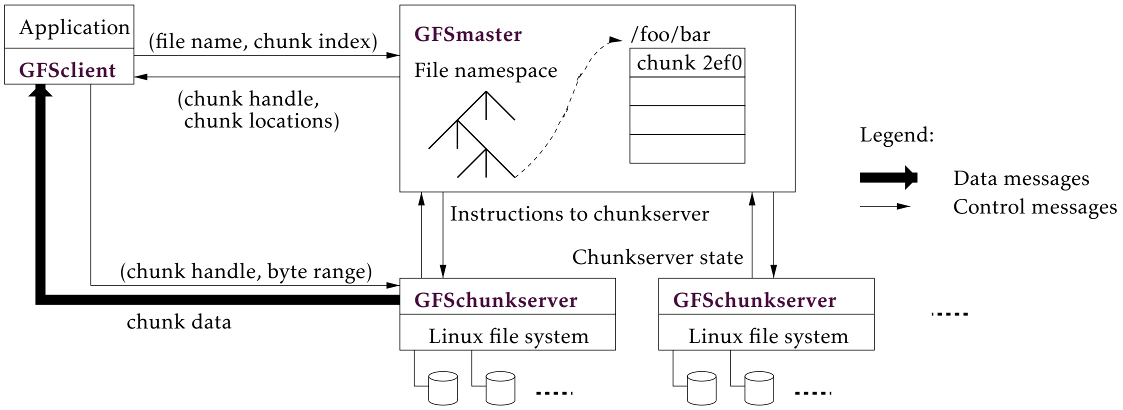

Figure 4: *The GFS architecture with the coordinating server, the GFS master, adopted from [2]*

single file to GFS [6].

Hadoop FS - implementation of GFS that has expanded over the years. drives Hadoop (map reduce) - high availibility module uses quorum - opt in consistency

Ceph

Subtree partitioning

References

[1] M Caporaloni 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] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. “The Google File System”. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP ’03. Bolton Landing, NY, USA: Association for Computing Machinery, 2003, pp. 29–43. ISBN: 1581137575. DOI: 10.1145/945445.945450. URL: https://doi.org/10.1145/945445.945450.

[3] Patrick Hunt et al. “{ZooKeeper}: Wait-free Coordination for Internet-scale Systems”. In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010.

[4] Flavio P Junqueira, Benjamin C Reed and Marco Serafini. “Zab: High-performance broadcast for primary-backup systems”. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2011, pp. 245–256.

[5] 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/.

[6] Marshall Kirk McKusick and Sean Quinlan. “GFS: Evolution on Fast-Forward: A Discussion between Kirk McKusick and Sean Quinlan about the Origin and Evolution of the Google File System”. In: *Queue* 7.7 (Aug. 2009), pp. 10–20. ISSN: 1542-7730. DOI: 10.1145/1594204.1594206. URL: https://doi.org/10.1145/1594204.1594206.

[7] Diego Ongaro and Ousterhout John. *In Search of an Understandable Consensus Algorithm (Extended Version)*. https://raft.github.io/. accessed 15-Feb-2022. 2014.

[8] S Shepler et al. *Network File System (NFS) version 4 Protocol*. RFC 3530. IEFT, Apr. 2003. URL: https://www.ietf.org/rfc/rfc3530.txt.

[9] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne. *Operating system conceps*. John Wiley & Sons, 2014.