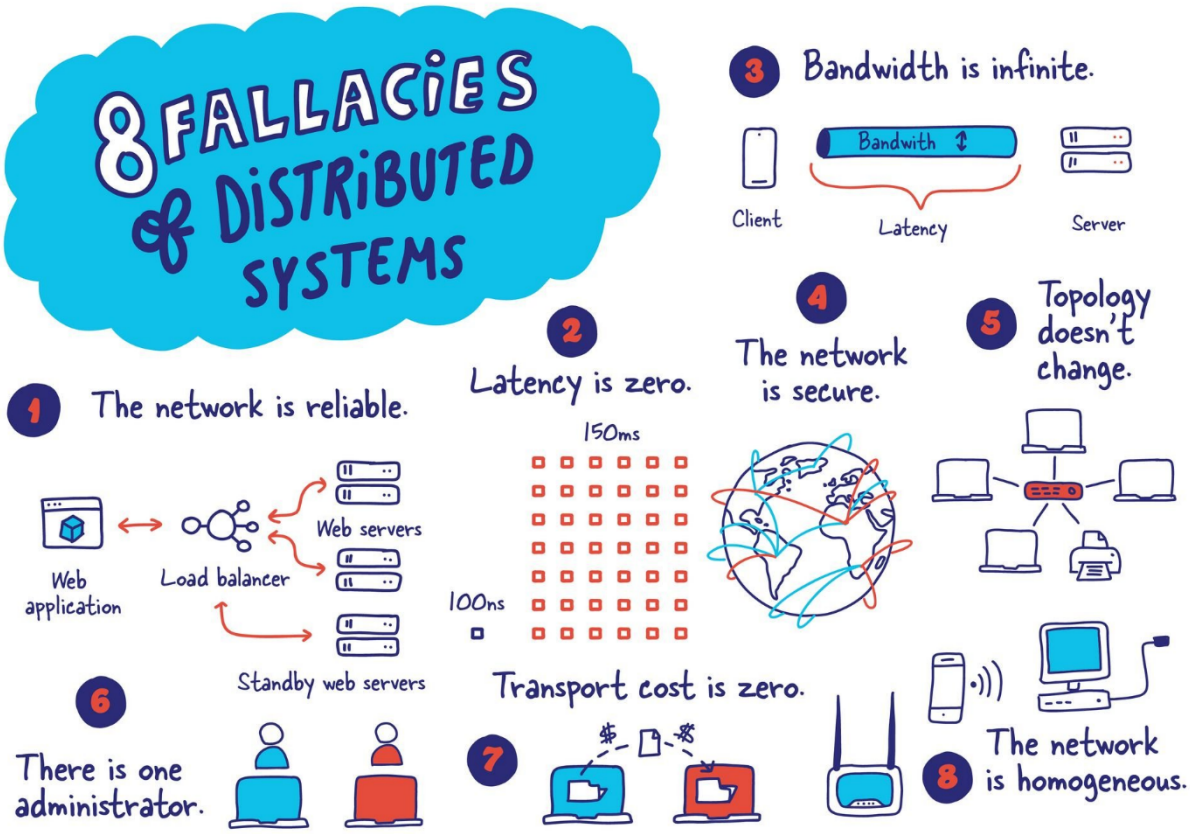
Distributed Computing Recap

Part A

Introduction

A **distributed system** is a collection of autonomous computing elements (**nodes**) that appear to its users as a **single coherent system**. To be coherent, the nodes need to collaborate (we need **synchronization**, we need to **manage group membership and authorizations** and we need to deal with **node failures**).



1. Both hardware and software can fail.
2. Latency is how much time it takes for data to move from one place to another.
3. Bandwidth is how much data can be transferred in a determined amount of time.
4. There might be malicious users trying to sniff communication packets.
5. New nodes can be added or removed from the network, and this can change many things, like the paths that packets need to travel and the latency.
6. Modern applications are composed of many services, which are controlled by more than one person.
7. It's related to the second fallacy, but this one is from the point of view of costs in terms of time and resources.
8. The network is heterogeneous: the computers used don't necessarily have the same configurations and hardware.

https://medium.com/geekculture/the-eight-fallacies-of-distributed-computing-44d766345ddb

https://architecturenotes.co/fallacies-of-distributed-systems/

There are three reasons for using distributed systems:

1. **Availability:** my application still works if one or more computers break down.
2. **Performance:** a single machine won't be able to handle the load or won't be fast enough.
3. **Decentralization:** avoid giving the control of the system to a single entity.

Transactions and Consensus

- **Transactions, ACID and CAP**

A **transaction** is an independent modification in a system that stores data (like a database or a file system). While they may change several parts of the system at once, we think about them as a single modification.

A classic set of properties to implement transactions are **ACID**:

- **Atomicity:** each transaction is treated as a single unit, that is it either succeeds or fails completely.
- **Consistency:** the system remains in a valid state.
- **Isolation:** even if transactions may be run concurrently, the system behaves as if they've been running sequentially (order).
- **Durability:** even in case of a system failure, the result of the (successful) transaction is not lost.

The **CAP theorem** states that, in a system that allows transactions, you cannot have all of **consistency**, **availability** and **partition tolerance** at the same time, only two of them.

- **Consistency:** every read receives the most recent write or an error.
- **Availability:** every request receives a non-error response.
- **Partition tolerance:** the system keeps working even if an arbitrary number of messages between the nodes of our distributed system is dropped.

In fact, let's assume that there is a system with two parts that don't communicate, G1 and G2. If a write happens in G1 and then a read happens in G2, the result of the write in G1 will not be available to G2, so one of these three situations happens:

- The system returns old data -> no consistency.
- The system returns an error -> no availability.
- The system doesn't reply -> no partition tolerance.

Distributed systems are consequently very often about **trade-offs**: either they will be offline until the network partition is resolved, or they will return an inconsistent and stale data. One example of system with inconsistent functionality is Git, because of the conflicts in the commits.

- **Consensus**

At some point, it's not manageable anymore to buy a bigger server that is able to do twice as many operations as the previous one but costs more than twice or a single-server solution simply can't handle the tasks anymore. Since a huge business can't afford losing money because there is a black-out or a flood, a desirable alternative would be to **divide the work between multiple servers**. These servers might have the

best performance/price ratio and they can be distributed geographically for better redundance in case of catastrophes, for example.

Two problems arise though:

- **Coordination:** the servers need to work **as if they were a single computer**.
- **Scalability:** make it so that **costs of coordination** are not huge.

Coordination can be achieved using ACID properties.

Deterministic state machines: Each machine has a **state** (which represents memory, registers, etc.) that determines its future behavior. They start from the same **starting state** and receiving an **input** (and eventually producing an output) changes the state **deterministically**. Hence, two machines are in the same state **if they have received the same input in the same order**.

The input is a **list (log) of commands**. Our distributed system can be based on servers that work like deterministic state machines, putting "write" commands that change the state in a **shared log**, and execute them on all machines and running "read" commands **locally**. If most of the workload of the system is made of "reads" (like querying a database or requesting a web page), which happens in many architectures, the system will scale well.

To implement a shared log, we need to consider the fail-stop scenario, in which messages can take an arbitrary time to arrive from one node to another (one-to-one messages only) if they arrive at all. Computers can **stop for an arbitrary time**, but they **don't lose data**. It's like using old Nokia phones to message.

This is not even the worst failure scenario: in fact, broken nodes may send any messages, maybe even acting maliciously. This is the kind of scenario taken into consideration in cryptocurrency and is called **byzantine consensus**.

In general, consensus is difficult to implement and use correctly. All machines responsible for consensus risk being a bottleneck (going at the speed of the slowest node because they are waiting for it). In common cases, a transaction takes a network round-trip to complete. If machines are all close, it's quick but what about correlated failures? Since nodes are close together, communication is faster because of low latency, but failure can affect more nodes contemporarily (example, floods). In addition, we need a trade-off between reliability and latency: in fact, a reliable system might implement systems of redundancy that increases latency and vice-versa.

- Paxos

Paxos goal is to achieve consensus considering a **shared log** on an asynchronous distributed system. It is a notably difficult algorithm, but basic Paxos to find consensus in **one log entry** is simpler, so that is the one we are going to focus now.

The idea behind Paxos is like having an old mobile phone and sending messages to agree with your friends where to go for dinner, following these steps:

1. Send to all your friends "Hey, what are you doing tonight?".
2. If most of your friends answered and nobody has plans, you propose "pizza".
3. If most of your friends agree, then it's decided.

The problem is that we need to consider that **failed nodes stop working** (someone's mobile has no connectivity, they may resume after getting signal again, but this doesn't guarantee that messages can't get lost or arrive after days), **but** fortunately we can assume that nobody loses their state, and everybody is honest and makes no mistake (no byzantine).

We want to **tolerate n failures**, so we need in total 2n+1 servers. That's because a decision is taken if the **majority n+1** wins (if n nodes failure, we still can receive an answer from the majority, that is equal to the remaining n+1 nodes). Since there can't be two majorities that doesn't intersect with each other, **at least one participant will see a conflict**.

Let's start with the successful scenario. Imagine having 3 servers (the minimum number of servers for Paxos, with n = 1), each identified by its number (1, 2 or 3). Each server can assume the role of being **proposer or acceptor**, or both. Let's assume that the first server is both a proposer and an acceptor while the other two remaining servers are acceptors. Paxos is divided into two phases: the **"prepare" phase** and the **"accept" phase**.

In the prepare phase, the first server sends a "prepare" message with its round identifier, a number that must be bigger than any other round identifier received by the acceptors previously, so a way to generate it is by using an incrementing counter (42 in the example). To guarantee that there isn't another proposer that has generated the same number simultaneously, we add the node identifier to make the round identifier unique (42.1 in the example).

After the message is sent, the acceptors will check if the round identifier is bigger than the ones they have received previously. If yes, they will save the round identifier in their memory and then they will send a "promise" back to the proposer. Otherwise, if it's not bigger, they will send a "negative acknowledgement", to notify the proposer that this is not going to work.

After that, if the majority has sent the promise back to the proposer, they will pass to the accept phase. In this majority, the first node that is both proposer and acceptor is included.

In the accept phase, the proposer will send another message "accept" (like, pizza for dinner) and all the nodes that accepts this propose will send a message with a proposal, called "accepted" (always after checking if the round identifier still is the biggest number). This include the proposer in this case, since the first node is also an acceptor. Again, if most of the nodes accepted the proposal, it's decided.

What could go wrong with this scheme?

- **Acceptor failure (in general):** If one acceptor fails during the prepare or accept phase, there is no problem if the majority still holds (node 1 and 2, for example, if the number 3 fails).
- **Proposer failure in "prepare" phase:** The trivial case is it fails before sending the first message. If it's after the first message, during the prepare phase, the other nodes send the promises, but the message "accept" never arrives. Since nothing happens, eventually another node should show up and starts its own version of Paxos as a proposer.
- **Proposer failure in "accept" phase:** If the proposer fails after sending the message "accept" for just one acceptor, let's say for example node 3. Since node 2 doesn't receive anything, they decide to start Paxos by becoming a proposer and finishes the job. This happens because the other nodes will answer the "prepare" message (43.2, calling for burgers, for example) with a "promise" message indicating that there was a previous "accept" message before (42.1). Node 2 will then just reassume the previous "accept" message (calling for pizza). It could also happen that node 2 is not notified about the existence of the former proposer and it just overwrites the old "accept" message (calling for burgers).
- **Two or more proposers:** The algorithm will never result in a wrong output, but in rare cases in could get stuck in an infinite loop of messages (livelock).

In the real world, we can use Paxos for leader election to have only one proposer at a time and use multi-Paxos to build a full log of decisions. This adds complexity, but it is possible to go faster (one round trip per transaction, after doing the prepare phase just once). It can also be used for cluster management. The only thing to keep in mind is that developing, testing, and debugging is hard.

- Raft

Raft is a distributed consensus algorithm as well, but its goal is to be easy to understand.

Nodes here are divided into three types:

- The leader
- The follower
- The candidate

The responsibility of the leader is to receive clients' requests and make sure that its own log is synchronized with the logs of the other nodes, the followers.

Time is divided into **terms**. In term 0, every node is a follower. After an election time out, one follower becomes a candidate and starts requesting votes. If it receives a vote from the majority of the nodes, it becomes the leader. If there are two candidates tied, they wait for the election timeout, and start requesting votes again. Since the election timeout are chosen randomly between 150ms and 300ms for each node, it is rare that two nodes start an election again at the same time and normally the first node to timeout is the one that wins.

After being elected, the leader notifies the other nodes, so the other candidates step down. To make sure that it's still alive during the term, the leader sends messages called "heartbeats" that doesn't contain any information. If the nodes don't receive any heartbeat, they can assume that the leader server is down and start a new leader election.

For log replication, when the leader receives an entry, it sends a message to the followers in the next heartbeat to make they append in their own logs the new entry. When the majority of the nodes has appended the new entry, the leader commits the entry and notifies in the next heartbeat that the entry was committed, so they can commit it as well.

Let's assume that the network is partitioned after leader election and the leader stays in a partition that doesn't have the majority of the nodes. If a new entry arrives to the leader, only the followers that are in the same partition will receive the append message. Since the leader will never receive an OK message back from the majority of the nodes, it cannot commit the entry.

On the other partition instead, there will be a new election after the timeout. Since the new elected leader will have the majority, it will commit the new entries. If the network recovers, the new leader will take over and the entries in its log that are not present in the nodes that were in the smaller partition will be written and the entries that were not committed because of the lack of majority will be deleted from the logs.

There are **three Log compaction strategies**:

- Leader-Initiated, Stored in Log
- Leader-Initiated, Stored Externally
- Independently-Initiated, Stored Externally

Before continuing with the discussion about consensus, it's worth clarifying the difference between **linearizability** and **serializability**. In the former case, the writes should appear to be instantaneous (corresponds to the consistency of the CAP theorem), so transactions are processed in the same order they are seen in the real world. In the latter case, it is not implied any kind of deterministic order (corresponds to the isolation of the ACID properties). It only guarantees that the execution of a set of transactions is equivalent to some serial execution (total ordering) of the transactions in terms of results.

- Zookeeper

Apache Zookeeper is an open-source **coordination service** for distributed applications which exposes a simple set of primitives to implement:

- Group memberships (add/remove workers/machines)
- Leader election
- Dynamic configuration
- Status monitoring
- Queuing, barriers, critical sections, locks…

Zookeeper is used to handle multiple outstanding requests and read-intensive workload. It is general, reliable, and easy to use. It has three building blocks:

- **Wait-free architecture:** No locks or other primitives that **stop** a server, no blocking in the implementation (which simplifies it) and no deadlocks. Needs an alternative solution to wait for conditions.
- **Ordering:** Writes are **linearizable** (they are executed in the order they are performed) while reads are **serializable** (you might read stale data). All operations on the same client will be serialized in FIFO order (queue style).
- **Change events:** Clients can request to be notified of changes (before seeing the result).

The ZooKeeper architecture consists of nodes called **znodes** organized in a hierarchical namespace, similar to a **distributed filesystem** (tree-like structure). Indeed, it works like a filesystem for small pieces of data (with the addition of the notification and minus the partial reads and writes). Znodes are like both files and directories, they have data and children.

It provides a simple **API** with the following methods:

- create(path, data, acl, flags)
- delete(path, expectedVersion)
- setData(path, data, expectedVersion)
- getData(path, watch)
- exists(path, watch)
- getChildren(path, watch)
- void sync()
- setACL(path, acl, expectedVersion)
- getACL(path)

The flags for the create methods are:

- **Ephemeral:** znode is deleted when creator fails
- **Sequence:** append a monotonically increasing counter

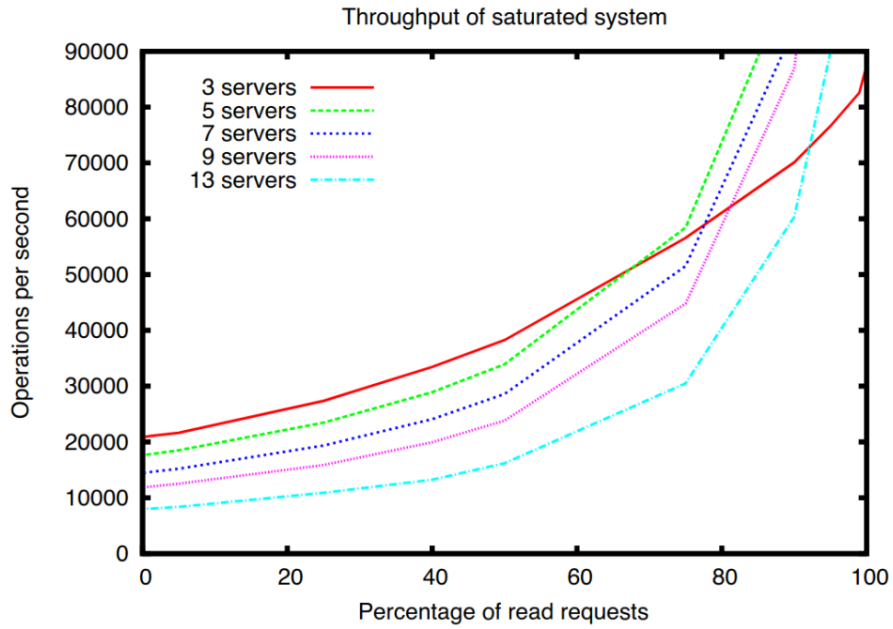
Now let's see which kind of operations we can do with these methods:

- **Configuration:** By using the method getData, a worker starts and get the configuration by using the path "/myApp/config". If it sets the watch parameter to "true", it will be notified in the case that admins change the configuration by using setData.

- **Group membership:** By using the method create, a worker starts and gets registers itself in the group, specifying the path "/myApp/workers/" + name of the worker and additional information. To list all the members of the group, the method getChildren is used for the path "/myApp/workers"
- **Leader Election:** If the method getData with the path "/myApp/workers/leader", is successful, it returns who's the current leader. Otherwise, the worker can propose itself as a candidate by using the method create, this time using as path "/myApp/workers/leader" and passing its name as additional information. If this operation is successful, the worker is the leader, otherwise it tries again. Thanks to the flag watch set to true, workers will know if the leader fails or changes.
- **Exclusive locks:** By using the method create with the path "myApp/locks/x-" and with the flag sequence set to true, it is possible to take a lock. The operation is successful if the id returned by the method create is the same as the id of the first child after calling getChildren on the same path (without the x-). Otherwise, check if the znode corresponding to the id returned with the method exists and set the watch flag to true to get the lock when its release. If it doesn't exist, return to getChildren.
- **Shared locks:** Similar to the previous case, but the path this time is "myApp/locks/s-" and the only thing to check after calling getChildren is that there isn't any previous exclusive lock "x-" before id. Otherwise, as before, use the method exists to wait for when this lock is released or go back to getChildren.

In terms of architecture, all servers have a full copy of the state in memory and Zookeeper uses a consensus protocol similar to Raft in which there is a leader that coordinates everything, and an update is committed when the majority of the servers have saved the change (always considering 2m+1 servers for tolerating m failures).

What is the ideal number of servers? You need to consider the cost (you need to pay for the machines and the energy), performance (depends on how many reads and writes) and failure resistance (the more machines you have, the more failures you can resist). In the graph below, it is possible to observe that if the system is heavily based on reads (right part of the graph), having more servers increases the number of operations. Conversely, on the left part of the graph, it is possible to notice that when the number of reads is small in comparison to the number of writes, it is better to have fewer servers (we need to update every log).



- Google Cloud Spanner

Google engineers have several datacenters distributed across the world which contains a sharded and replicated database. They wanted linearization, because, for example, if you exclude someone from your followers, you don't want that person to see what you are publishing.

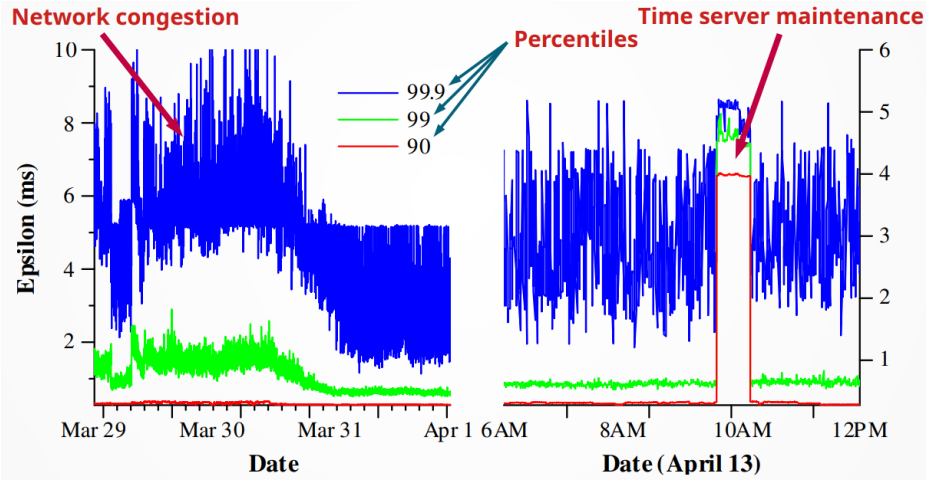
Using Paxos or Raft to build a log of all transactions is unfeasible given that we would need to update all servers. Using exclusive locks for writes/shared locks for read would solve our problem, but it is unreasonable for huge systems since when there is a write nobody can read until the lock is released. The solution is to use Paxos locally and then resort to **clocks** (that are more reliable than CPUs).

Pieces of information are annotated with time, indicating when they got in the system and when they eventually get deleted. To do this, we need a **super-precise** clock, which is difficult considering Internet latency. Spanner will **wait** to drive uncertainties in the clock down.

The idea is to use **bounded uncertainty** (TrueTime.now() returns an interval [earliest, latest] in which the real time is included). Each datacenter has a set of **time master machines** (regular ones equipped with GPS and Armageddon masters with atomic clocks). After the client request the current time to the time master, it must consider the time that the information takes to travel, as in the NTP protocol. After repeating the measurements to get some statistics and getting an estimate of the time using an interval.

Since one of the time masters could be wrong, we need to select the subinterval most consistent with the n intervals acquired using the **intersection algorithm**.

In terms of errors, we see that using high percentiles instead of just the median shows that the system most of the times works well, minimizing errors related to the timing synchronization.



The **data model** is a key-value store and nodes are responsible of a key in **multiple continents** (Paxos to get consensus). This allows asking the value **at a given moment in time** with the use SQL-like semantics added afterwards.

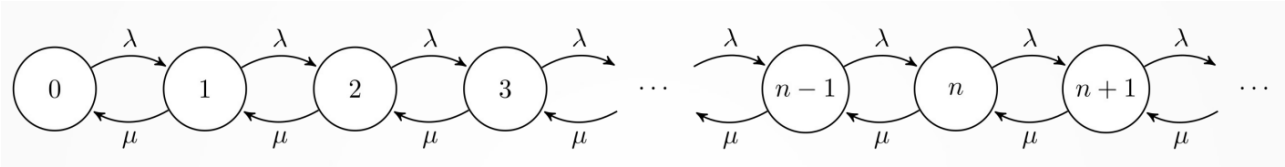
The timestamp used is any moment when the lock is acquired by the transaction. We assign to the lock time the latest time possible of the interval returned by TrueTime and the unlock time the earliest time possible of the interval returned by calling TrueTime again. This guarantees that the actual time of acquisition and releasing of the lock is coherent in terms of time progression.

Using this strategy, if two non-conflicting transactions acquires the lock, there is no problem since the acquisition and releasing times for the first will be smaller than the ones for the second, but even in cases of conflicting transactions (run contemporary), the problem doesn't arise since we can't get a lock if it's already in use (their intervals will be disjoint).

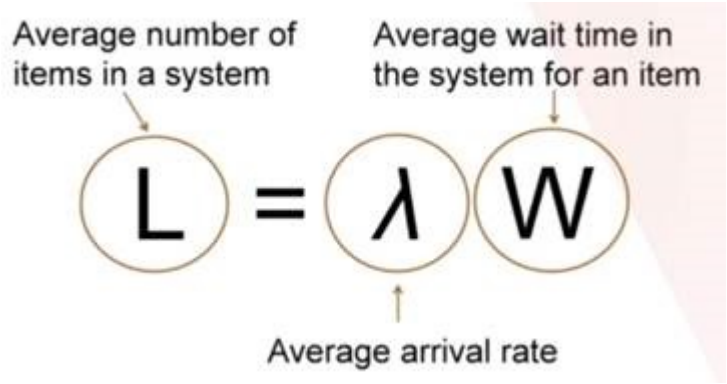
Queuing and Scheduling

M/M/1 model: A single server and a queue of jobs it must serve, in FIFO order. Jobs arrive and are served in **memoryless** fashion (the probability of the arrival or serving of a job never changes). Real systems are not like this, but some of the insight does apply to real-world use cases. How does this model behaves in the long run?

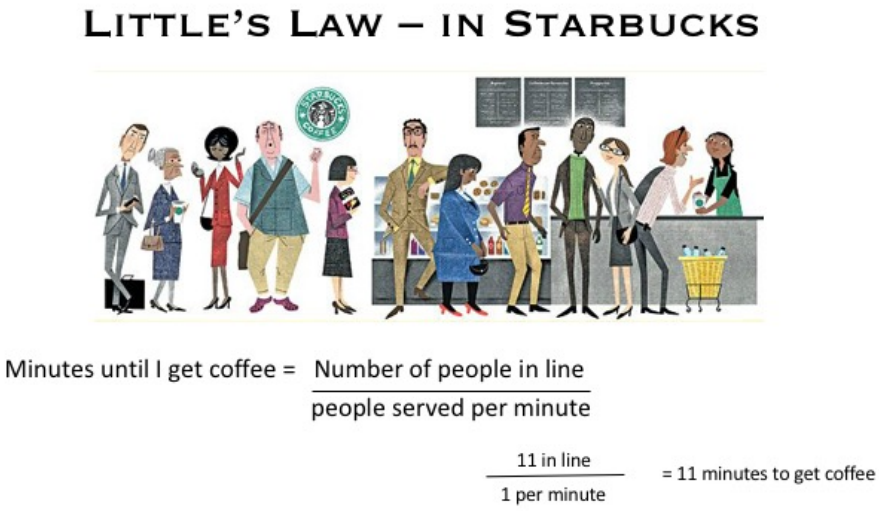
To be an equilibrium probability distribution, the arrival rate lambda needs to be smaller than the leave rate, otherwise the number of elements in the queue will keep growing, and the probability of moving to one state to another is the same going “forwards” and “backwards”.



We can use **Little’s Law** to calculate the **average length of the queue**. The idea is to multiply the arrival rate lambda times the average wait time for each job. For example, if in a supermarket the time to wait is 10 minutes and the arrival time is 1 per minute, I know there are 10 people in the queue.



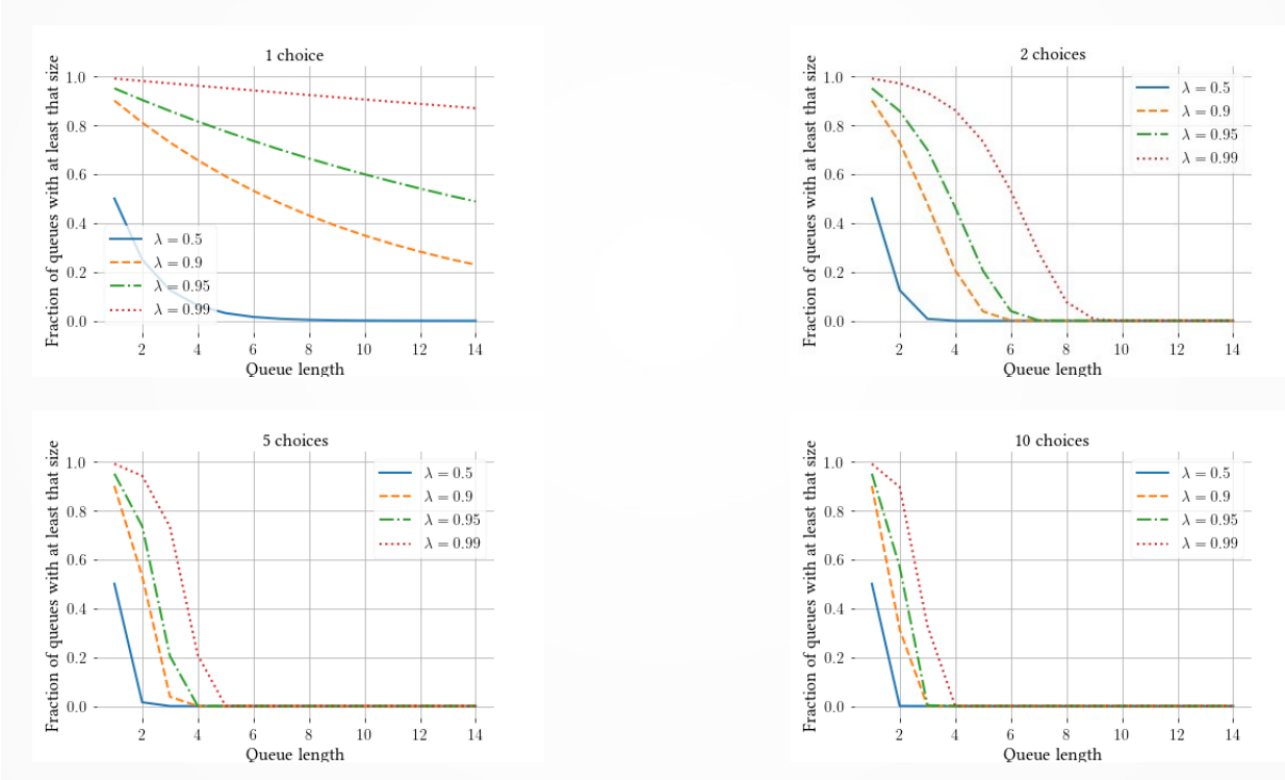
An inverse formula is calculating the **amount of time a typical job will wait before being served** by diving the length of the queue by the average arrival rate.



Multi-Server version: A load balancer divide the load between n servers. If we assign the jobs to the servers randomly, you can handle a load of nλ with the performance of a single server having load λ. Assigning jobs to the least loaded server though is a better strategy. By using the **supermarket queuing**, rather than

querying n servers to find out their queue length, just ask a small number d of them and assign to the shortest queue.

Below, we can see that if lambda is high, the queues are longer (because the arrival rate is high many jobs arrive at once), and a way to drop it is by using more servers to choose from.



Beyond FIFO:

- **Preemptive policies:** In the real world, you very often have many **very small jobs** and a **few extremely large ones,** so in this case the memoryless property doesn't apply. You can **preempt** running jobs (pause them and resume them later) using, for example, round-robin scheduling, which run each job for a given timeslot (processing speed 1/n for n jobs in the queue) and then pass the turn to another one (often giving priority to certain jobs).
- **Size-based Scheduling:** If you know for how long a job will be running (its size), the optimal policy is Shortest Remaining Processing Time (SRPT), which always serves the job needing the least work to complete (theoretical risk of starvation for large jobs, but not relevant in practice). SRPT may behave catastrophically if jobs' size information is incorrect because if large jobs are underestimated, their "remaining processing time" goes below 0 and "blocks" the system until they are finished (e.g., if the system estimates 5s instead of 10s, it will run it for 10s and mark -5s), which is problematic with very different job sizes. The solution is to first schedule the jobs with smallest estimated size instead of considering the remaining times, assuming estimation error is proportional to real size.

To conclude, using mathematical models is good because it is rigorous, but needs simplification. Conversely, using real systems is good for verifying how things behave in practice, but overly focus on implementation details, is expensive and limited. Using simulations as we did is a trade-off between the too, being scalable and cheap, but not as precise.

In general, the main idea is applying the supermarket technique and considering if it is useful to use FIFO or an alternative (round-robin or size-based if you have half-estimates of the jobs' sizes, prioritizing shortest jobs and taking into consideration malicious behavior if possible).

https://yorkesoftware.com/tag/littles-law/

https://opexlearning.com/resources/cycle-time-reduction-littles-law/9023/

Erasure Coding

Problem: I can store data on N different servers and I want to be able to recover my data even if M servers fail and lose my data. How to minimize the cost?

Before getting to the possible solutions, let's define the concept of **redundancy** (it's the ratio between the total amount of data store and the original amount of data).

- **Replication (Trivial solution):** To safeguard myself against M server failures, I put a

  copy of all my data on each of N = M+1 servers. The redundancy in this case is M+1. For example, if I want to store 100GB of data and want to be safe if 2 servers die, I need 3 servers, each containing 100GB of data, because in this case if 2 of them fail I still have the third one. In this case, I store 300GB of data considering the 3 copies and 100GB is the actual data (300/100 = 3 = M + 1, since M = 2).

- **Parity (N = 3, M = 1):** I split my data in 2 blocks B0 and B1 then I create a **parity** redundant block BR, that is the result of the xor between B0 and B1 (we can use xor because the blocks contain binary information). Thanks to the properties of the xor, if we loose B1, we can get it back by calculating Br xor B0, and vice versa if we lose B0. For example, if B1 and B2 store each 50GB, BR will store 50GB as well, so the total amount of data is 150GB and the actual data is 100GB, so the redundancy is 150/100 = 1.5.

- **Parity (Any N, M = 1):** The same concept explained before can be extended with more than two blocks always using an extra block BR for parity. For example, if we have 6 blocks containing each 20GB, we need in total 120GB of data (the extra 20GB is for BR) for storing 100GB of actual data, so the redundancy is 1.2 in this case.

- **Erasure coding magic (Any N and M):** I encode my data in N blocks, each of size 1/Kth of the original data, where K=N-M. I can decode any K of those blocks to recover my original data. Redundancy in this case is N/K. For example, if my actual data is 100GB, I'll have six blocks of 25GB, so 150GB in total and 1.5 redundancy.

- **Linear Oversampling (Any N, M = 2):** A line y = ax + b is defined by two points (x1, y1) and (x2, y2). This means that we can send a message a = ? and b = ? by only sending y1 and y2 (considering x1 and x2 predetermined), since we can solve the linear system ax1 + b = y1 and ax2 + b = y2.

- **Polynomial Oversampling (Any N and M):** We can extend the above concept to a polynomial of degree K-1 which needs K points to be identified (where K = N - M). Here the idea is to find the coefficients from c_0 to c_{k-1} (a and b before) of the expression $y = c0 + c1x + c2x^2 + \ldots + c_{k-1}$ by using the known pairs (x, y), which is easy since the known values for the x parts makes the non-linear part disappear. Again, the idea is to solve a linear system, the only problem is that the numbers in the encoded message (the y parts) will be bigger than those in the original one (the coefficients). A way to solve this is by using Galois fields, which are finite and preserve operations like the sum, multiplication, and multiplicative inverse (thanks to choosing a prime number).

- **Other approaches:** Approaches that "waste" a bit of space compared to the "perfect" result but give you great properties. Fountain codes, in which you don't choose N and generate as many redundant as you want (such as a stream), and regenerating codes, in which if you lose one or a few blocks you don't need the original plaintext to recreate them, just download a few encoded blocks and work from them.

Peer-to-Peer Systems

- Tor

More than 90% of the websites do some form of **tracking** (despite the GDPR law and user consent) to get cross-site information and mobile apps probably track even more, since it is difficult to investigate, and they have more access to sensors. The main player is the advertising industry, but more covert ones could be government agencies and malicious entities.

The main argument against **complete privacy** is crime and terrorism, but some arguments for it is that criminals already have access to anonymity (stole phones and compromised machines) and it can be useful to protect people (such as whistleblowers, activists, and journalists) from being tracked by totalitarian governments and corporations or being psychologically manipulated.

A solution to be anonymous on the Internet is **Tor**, a project initially funded by the USA government and then picked up by a non-profit organization for digital rights, which goal was protecting intelligence communication online by using **onion routing**, which is based on **layers of encryption** (like those of an onion)

In onion routing, a message is sent through n routers chosen randomly each time to create a path and is intended to a destination server. Each router "peels" one layer of encryption and sends the rest to the next step, until the message finally gets sent by the last node to the destination, after removing all the crypto layers. No router knows **both the source and the destination**. Some useful terms are guard, relay and exit node (first, middle, and last node, respectively, keeping in mind that there could be more than one relay node) and directory server (the servers containing the IP addresses of the Tor nodes, which are public).

Some countries block access to known Tor nodes, so **non-public bridges** exist to allow people to use Tor anyway. Bridges can be discovered (and censored) with a full scan of all the IPv4 addresses or with deep packet inspection (DPI). A countermeasure to the last measure is to use **obfuscation** (pluggable transports): together with the bridge address you get a secret and some protocols hide your protocol to DPI.

It works under two assumptions:

- All the routers you choose shouldn't be owned by the same attacker.
- Attackers can't see your traffic from both guard and exit nodes (otherwise it's easy to correlate your traffic).

Tor doesn't support UDP connections, so it uses SOCKS as a proxy, which is a protocol that allows encapsulating TCP connections. For this reason, using BitTorrent over Tor is not a good idea, since it uses UDP.

The **Tor Browser** is a hardened Firefox designed to look identical for all users and never exit from the proxy. Using it on your everyday browser is not a great idea:

- **Cookies:** website can sync cookies to correlate your visits on different websites.
- **Fingerprint:** specific information on your hardware/software

Onion Services (Dark Net): The server picks some **introduction points** (routers) and builds circuits to them advertising its service (onion address) to the database. The client requests more information about the server's service (including its introduction points) and sets up a **rendezvous point** (router). It sends a message listing the rendezvous point, a one-time secret (like a cookie that the server can recognize) and asks an introduction point to the deliver it to the server. The server answers to the rendezvous point providing a one-time secret as well. If the two secrets match, the rendezvous point acts like an intermediate for the communication circuit, connecting the client and the server, like any other onion router.

In **numbers**, Tor has around 5 million users and has peaks probably caused by technical or political issues. 1/5 of the total usage is for illicit content. The security of Tor is not perfect, but it is good enough in most cases (people were caught because of mistakes, not attacking Tor).

Regarding attacks, a technique to identify which website a user is looking at by looking at the sizes and timing of encrypted packages, but Tor uses messages of a fixed 512-byte size (cells), which together with higher latencies makes this technique difficult. The real-world fingerprinting problem is more difficult because websites are a lot and change frequently, but since darknet sites are less in might be more relevant in this case.

- Search in P2P Systems

Peer-to-Peer systems are **distributed and decentralized** applications (hence fault tolerant, since an app owned by a single organization may be shut down) where nodes (**peers**) play "equal roles". They are self-organized and adaptive, censorship-resistant (as seen with Tor) and uses resources that would be wasted otherwise.

Problem: In a system with potentially millions of peers, how to find a given piece of content? We need to balance decentralization with performance.

Napster (1999): Napster was a file-sharing system used mostly for MP3s which was legally tricky: uploading copyrighted content was illegal, but what about just telling where it was? It used a **central** index server, to which users uploaded information about their songs, which arguably caused it demise in 2001 (there was a central server to shut down and a company to sue), so people started working on **completely decentralized applications** by creating **overlay networks** (networks on top of other networks, such as P2P over TCP/IP).

Gnutella (2000): It was the first decentralized P2P file-sharing network. It used overlay network where each node is connected to a few others (5 by default). On initial startup, each node contacts some services ("caches") to get some nodes to connect with (this is called **bootstrap**). If a node is "full" of connections, it will forward the connection request to its neighbors and new nodes discovered will be saved for the next sessions.

Initially, it used **flooding** for searching. For a new query, a time-to-live TTL (max 7) is set and forward to all neighbors, each of them will decrease the TTL and forward the query to its neighbors as well and so on, until the TTL reaches zero. It worked to some extent, but there were many redundant messages that increased the possibility of overloading weakest machines.

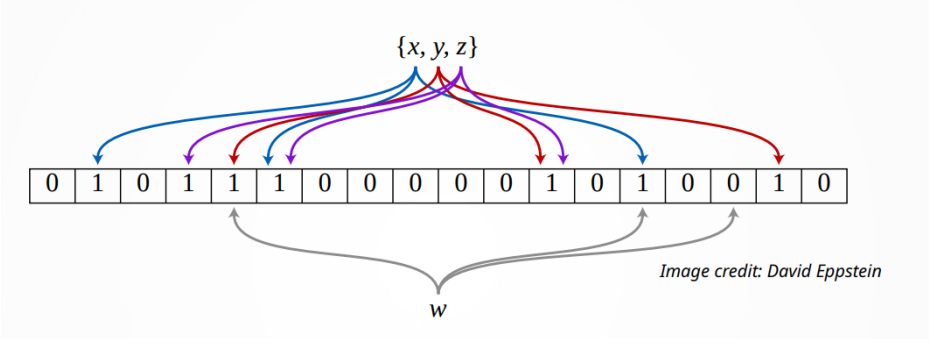
A solution was using a design pattern that distinguished between **leaf nodes** and **ultrapeers.** Nodes with bandwidth and stability could get upgraded to ultrapeers and connect to 32+ other ultrapeers, while leaf nodes connect to 3 ultrapeers. The search in this case works as follow: ultrapeers aggregate their **QRTs** (query routing tables, a representation of the set of files they have) and those of all their leaves and sends the results to all their neighbors. Queries get sent to a peer only if they have a hope of having the requested file. These modifications greatly improved scalability and TTL was lowered to 4.

QRTs should represent a set of keywords as small as possible where we can have false positives, but **no false negatives** (e.g., if my query is "foo bar" and I receive a positive answer because there is a file with "foo" and another with "bar" but none "foo bar" there is no problem, but it is a problem if a node containing "foo bar" is ignored). For implementing them, we can use **bloom filters**, a data structure for representing sets, giving two methods:

- add(x): add an element to the set
- test(x): tell me if x is in the set

There is the possibility for false positives and no way to retrieve the original elements, but on the other hand it is very compact. In the example below, since x, y and z are stored on the QRT, the three

corresponding bits are 1. If we search for w, there is one bit set to 0, so we can be certain that w is not on the QRT because there are no false negative. If we searched for another value, k, that had the three corresponding bits set to 1 but was not actually present on the QRT, there is no problem since it is a false positive.

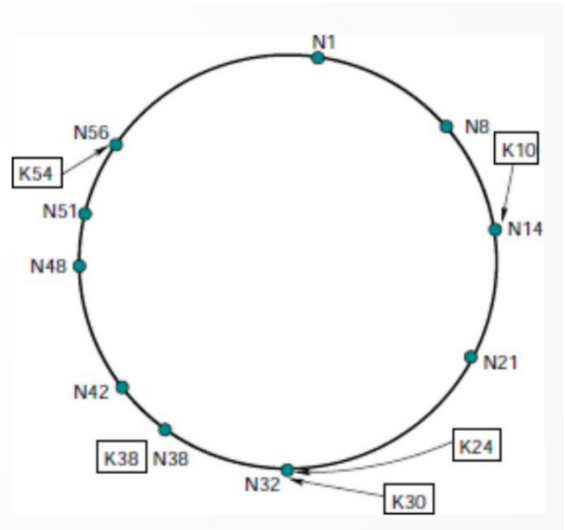


It has several applications, for example, in databases we can keep a bloom filter on RAM (cache) to spare a disk access if we know an item is not there, and for web caches, in order to avoid storing data requested only once.

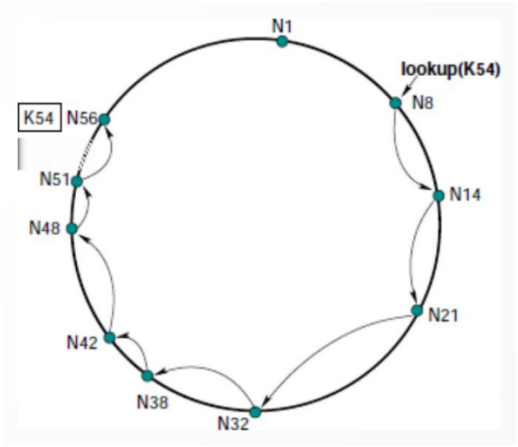
A better idea than using superpeers is using a decentralized system giving efficient key-value lookups called Distributed Hash Table (DHT). It is a structured peer-to-peer overlay in which we **choose who connects to whom** and we use that freedom **to obtain efficient routing**.

For redundancy, more than one peer handles a portion of the hash table. This allows **consistent hashing**, that is, adding or removing peers has a small impact on resource allocation (i.e., which data a peer stores), which is perfect to handle **churn** (nodes arriving and leaving all the time). Item x will be stored on node corresponding to address h(x).

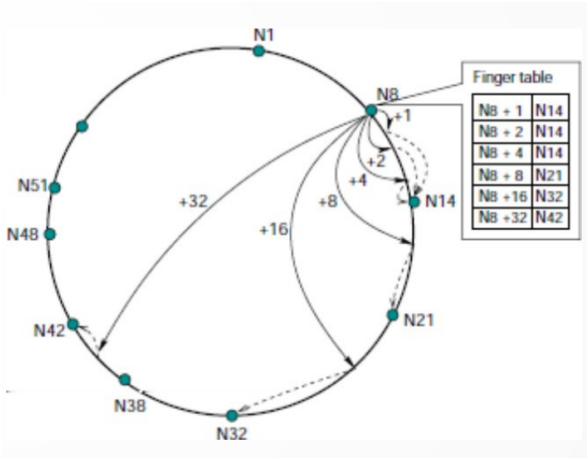
We can implement DHT by using the **chord ring**, in which each node gets a random identifier within a certain range (in the example [0, 63]) and get in the ring with a link to predecessor and follower. Item x gets inserted at the first peer with hash greater than h(x). In the example below, (K54 is assigned to N56 because the predecessor N51 is too small, 51 < 54). For redundancy, it is also stored in a few predecessors.



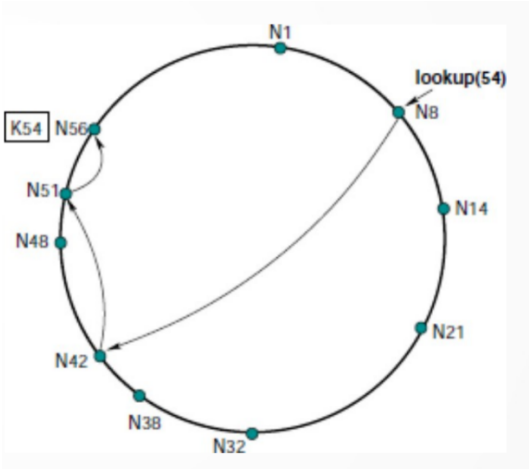
The **slow version** of the lookup when using the chord ring is following successor link until you get to the node responsible for a given key. This system has complexity O(n) (n if the node I'm searching is the last one) and risks breaking if any node in the middle disappears.



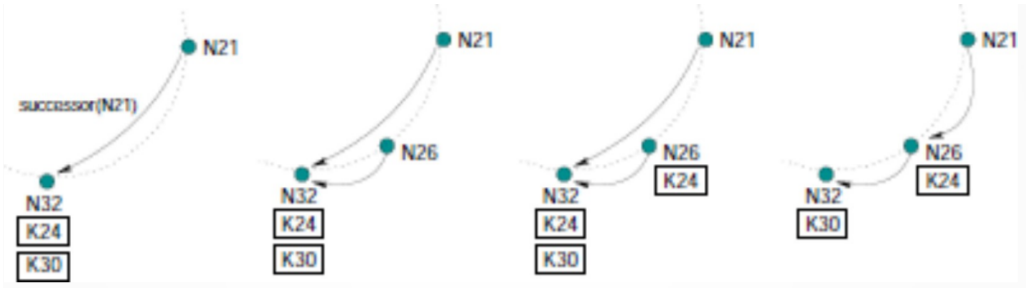
The **accelerated lookup** consists of each node keeping shortcuts to destinations at exponential distance to get there faster (for example, node 8 points to node 14 when summing to 8 the values 1, 2 and 4, to node 21 when summing to 8 the value 8, and so on).



A similar approach but faster approach, called **Greedy** routing, consists of at each step we follow the finger closest (but before) the destination (complexity O(log n)).

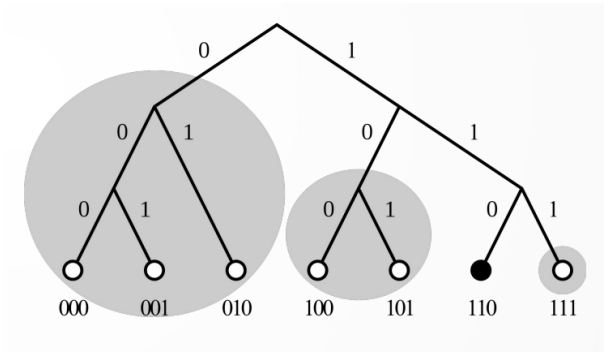


To add a new node to the chord, the only things to do is a lookup to see where to position it and then exchanging data with the predecessor and successor (in the example, we put N26 between N21 and N32, and assign K24 to N26 instead of N32 because 26 is closest to 24, but still larger).



In terms of fault tolerance, to avoid breaking the successor chain when peers leave the system, nodes keep a list of more than one successor. Still, if all of them leave the system, the ring can be broken. To avoid this type of situation, a periodic stabilization procedure maintains the links: predecessors and successors get pinged, fingers are re-queried.

The most used DHT in practice (for example, in BitTorrent) though is Kademlia, which uses a logarithmic number of steps to get to destination (as shown below). An advantage is that since links are symmetrical, you can exploit information about them when they reach you.



- BitTorrent

BitTorrent is a **file distribution system** in which is **not possible to search** for files. A .torrent file contains metadata about your file (name, size, hashing information, the URL of a **tracker** if you are not using DHT).

A tracker is a machine that helps node discover each other. A **seed** is a node with a full copy of the file, just uploading it, while a **downloader** is a node that has not finished downloading a file yet. When downloaders complete a download, they can stay and become seeds.

Files are cut in **pieces** (256KB by default) and hashes of each piece are included in the .torrent file. Nodes contact each other, asking which pieces they have, and a node doesn't report having a piece until it verifies if the hash matches. We always try to get the rarest piece in the system first (since it will be the one that will be most difficult to find later and to guarantee that copies will stay around), even though exceptionally we can get a random first piece, to get something to upload as soon as possible.

Among all the uploads open, most are **choked** (uploaders don't send data through them). By default, 4 connections are unchoked: 1 is the **optimistic unchoke**, in which every 30s give an upload slot to a random node, and the other 3 go to the peers that are sending data faster, which are recompensated with fast downloads. This method incentives cooperation (otherwise, why would a node upload instead of just downloading?).

- Eventual Consistency

We've seen CP systems (if Paxos/Raft are partitioned, the minority will stop working). Now we'll delve in AP systems.

NoSQL databases came before SQL.

In the 1970s, there were two world views:

- Relational (top-down approach): It is an easy-to-grasp abstraction and clean model in which one API does it all (SQL). SQL is a declarative language where the user doesn't need to worry about optimizations and there are two kinds of developers, the DB authors and SQL programmers. Data outlasts implementation in this case and ACID transactions are considered. In general, do one important thing, do it well.
- UNIX (Bottom-up): It consists of few simple mechanisms and compose tools. The bottom-up approach consists of adding new modules for new functionalities, which allows flexibility for growing the system. There is only one kind of programmer.

In the 1990s, to build a search engine and a proxy cache people used custom servers on top of file systems because they were faster (DBMS's features cost performance). ACID vs. BASE (Basically Available, Soft State, Eventual Consistency) battle, even though BASE wasn't initially well-received since people liked ACID. After the CAP theorem though, in the 2000s, eventually consistent systems started to get used.

In partition mode, we need to understand if we can allow commits or any impact on the outer world or give output. And after the partition recovery, how can we get back to a consistent state? By letting the last writer wins or using rules that depend on what has been done?

ATMs do keep operating when isolated from the network in partition mode (for this reason, there are withdrawal limits of amount of money). After the partition recovery, we need to detect errors (balance below zero) and compensate (overdraft penalty).

Three key issues to address in eventually consistent systems:

- Detect partitions.
- Define how "partition mode" works.
- Define how to do recovery.

The real world is eventually consistent, because there are "consistency rules" (laws, contracts, ...) and if you see problems (inconsistency detection), you compensate for it (money, ...).

One example of eventually consistent system is **Amazon Dynamo**, which is used in handling shopping carts at Amazon and was born as a key-value store (becoming later a DB). Since availability affects income, it is chosen over consistency (only 2.5 minutes of unavailability in a year). In fact, you "always write", that is, add an item to your shopping cart, even in partition mode.

Chord in a datacenter (nodes are servers) using consistent hashing (adding/removing one node at a time is cheap). It is completely decentralized, and each item is replicated in the N nodes "after" a given key in the ring. Those N nodes are called "preference list" and replication guarantees durability.

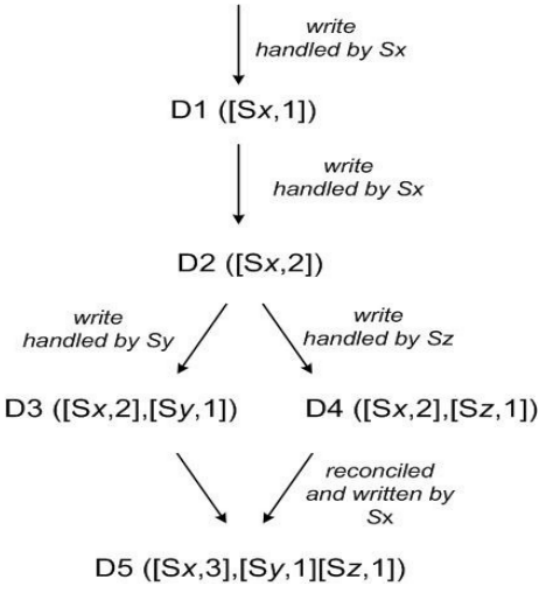
Unlike Chord, every node knows the full partition table, so it takes just one hop to get to any piece of data. The routing table update and **failure detections** are done via a **gossiping** algorithm, in which every node periodically exchanges information with a random set of other nodes. Optionally, the client knows the routing table as well and data can be directly asked to the node having it.

For faster partitioning, split the hashing space in Q partitions. They get distributed equally between nodes. When nodes join, they "steal" partitions from other nodes to maintain the equilibrium, and when they leave, they are redistributed to other nodes. Transferring partitions **doesn't require random disk accesses**.

The API consists of two methods:

- get(key) -> [value], version_info: it returns a **list** of values, which may be more than one in case of inconsistencies. In this case, they will be handled by the client. version_info is passed to the subsequent put to solve inconsistencies, being null if something is created from scratch.
- put(key, value, version_info)

version_info gets a counter value for each machine they have passed through (**vector clock**). One copy supersedes another if counters are not smaller for each machine. Otherwise, they're independent and we ask the client what to do.



Merkle trees are used to compare data between nodes that store replicas of a partition. The tree is constructed by recursively hashing pairs of data (or hash values) until a single hash, known as the root hash, is obtained. Each leaf node represents a specific piece of data (or a hash of that data), and each non-leaf node is the hash of its children. When comparing data between nodes that store replicas of a partition, the Merkle tree structure is utilized. The nodes can compare the root hash of their respective Merkle trees. If the root hashes match, it indicates that the entire dataset is identical, otherwise it means there is a difference in at least one part of the dataset, so we compare the children to find out which half is different, and so on recursively. It is a fast way to spot differences and reconcile them.

In terms of **client-side reconciliation**, if everything else fails, the client is presented with more than one return value (rare). In the Amazon cart policy, it means that in doubt it leaves stuff it in the cart.

Other optimizations are:

- Buffered writes: wait for a few writes to be committed before writing to the disk (Performance/ consistency tradeoff)
- Throttling background operations (slow down gossip/maintenance when many requests are around)
- Let coordinate read/writes to nodes who are responding fastest (additional load balancing)

Sloppy quorum consists of three configurable parameters:

- N: number of copies of each piece of data.
- R: number of reads to get a successful read (Low R, fast read, high R, consistent)
- W: number of writes to get a successful write (Low W, fast write, high W, consistent)

If **R+W>N**, it is a sort of consensus algorithms, guaranteeing high consistent **except in case of failure**. In this case, when machines go offline, it's considered transient, since permanent addition or removal is an administrator action. Reads and writes spill over to the first machine in the ring after the currently unavailable N machines that should handle them by default. When the machine comes back online, updates are reported to it.

Facebook's **Cassandra**, now handled by Apache foundation, uses a very similar architecture. It uses Zookeeper for routing table, seeds and to elect a leader and coordinate it for rack-aware & datacenter-aware data placement. It uses timestamps instead of vector clocks, and the latest wins. Lightly loaded nodes get "migrated" on the ring.

Introduction to Large-Scale Data Processing

- The MapReduce Paradigm

Big data is characterized by the 3Vs (volume, velocity, and variety) and it lead to the realization that **data often counts more than algorithms**. For that reason, the **MapReduce** programming model was invented, inspired by functional programming and bulk synchronous parallelism. Let's see some of its principles:

Its execution framework is for large-scale data processing running on "commodity hardware", so medium-range servers because high-end servers cost grows more than linearly with performance and often a server big enough simply doesn't exist.

In many workloads, **disk I/O is the bottleneck,** and we want to read from several disks at the same time. A way to alleviate this problem is by using a **shared nothing architecture** in which we **minimize sharing** by synchronizing the disks only when we need to, **to avoid latencies** and implementation bugs. In this case, we have independent entities, with **no common state**.

When you have a cluster that's big enough, failures are the **norm** (hardware/software failure, natural disasters, cascading failures when the failure of a service makes another unavailable…), but most failures are transient (data can be eventually recovered).

In high-performance computing, there is a distinction between performance and storage nodes. For CPU-intensive tasks, the computation is the bottleneck, but for data-intensive tasks the bottleneck is generally the network or the disks. In the second case, we want to process data close to the disks where they reside, so **distributed filesystems** are necessary, and they need to **enable local processing**.

Since data is too large to fit in memory, it's stored on disks. There's a big advantage in **organizing computation for sequential reads**, because rewriting all the records takes just a few hours instead of one month to update only 1% of the records with random access.

MapReduce is used for batch processing involving (mostly) full scans of a dataset. The data is collected elsewhere and then copied to a distributed filesystem. One example is compute PageRank, a score for the "reputation" of each page on the Web.

In two dimensions, the **scalability goals** are:

- **Data:** if we double the data size, the same algorithm should ideally take around twice as much the time.
- **Resources:** if we double the cluster size, the same algorithm should ideally run in around half the time.

Embarrassingly parallel problems are computational tasks that are inherently well-suited for parallelization because they can be easily divided into independent sub-problems that require little or no coordination or communication between them. These problems are particularly advantageous in shared-nothing computing environments.

Now, let's dive more into the MapReduce programming model:

There are two higher-order functions (functions that accept functions as arguments):

- **Map:** Apply the function m to each element of the sequence taken as input, producing a new sequence as output (map(neg, [4, -1, 3]) = [-4, 1, -3])
- **Reduce:** Given a list l with n elements and an initial value v0, computes the function r applied to v0 and the first element of the list to get v1, then applies r to v1 and the second element of the list and so on (e.g., l = [1, 2, 3], sum(l)=reduce(add, 0, l) = 6).

On the MapReduce framework, there are three phases:

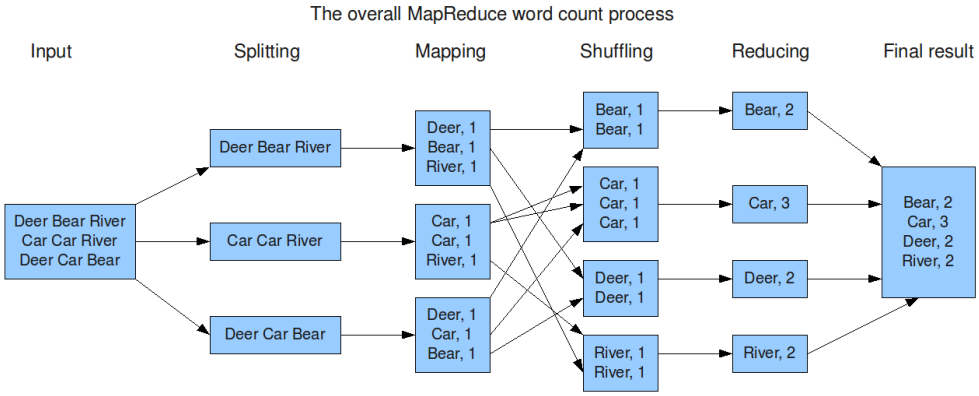
1. **Map Phase:** Processes data **where it's read** (data locality). Firstly, filter what's not needed to not waste network bandwidth sending it afterwards. Then **transform data**, which unlike the functional map, this always creates key/value pairs. Since it works in an **embarrassingly parallel** manner, each "fragment" of input determines its own output.
2. **Shuffle Phase:** Data gets **grouped by key**, so that we get a sequence of all values **mapped to the same key**. Handled by the execution framework (e.g., Hadoop), so programmers don't have to do anything, although there are optimizations possibly visible to the users. Data gets **moved on the network** and if it is well distributed along keys, work is well distributed between machines.
3. **Reduce phase:** An **aggregation operation**, defined by the user, is performed on all elements having the same key. The output is written on the distributed filesystem and **can be an input to a further map-reduce step**.

A way to reduce the amount of data before sending it over the network is to use **combiners**, which are "mini-reducers" run on mapper machines to pre-aggregate data. In Hadoop they're not guaranteed to be run, so the algorithm must be correct without them.

In general, we want to know what we can do **efficiently** with MapReduce by finding scalable solutions, which is nontrivial. In fact, many algorithms require **multiple rounds** of MapReduce.

For example, let's consider the problem of building a nxn co-occurrence matrix M, where n is the number of words and a cell mij contains the number of times word wi occurs in the same context of wj (NLP). M has size n^2, so it can become very big quickly (e.g., in English there are hundreds of thousands of words). Since most of those cells will anyway have a value of 0, we can use the **pairs approach** that utilizes **complex keys**: when the mapper encounters w1 close to w2, it will emit the ((w1,w2), 1) pair, meaning it has found the (w1,w2) pair once. We can also use the **stripes approach**, in which the mapper will associate to the key a mapping to all the words corresponding non-empty columns in a matrix row and then the combiner and the reducer will aggregate each of the stripes.

Another example is word count:



- Apache Hadoop

Apache Hadop is a free-open source based on Java handled by the Apache Foundation. It is a large ecosystem, so we'll focus on the parts that deal with MapReduce.

The principles of **HDFS (Hadoop distributed file system)** are:

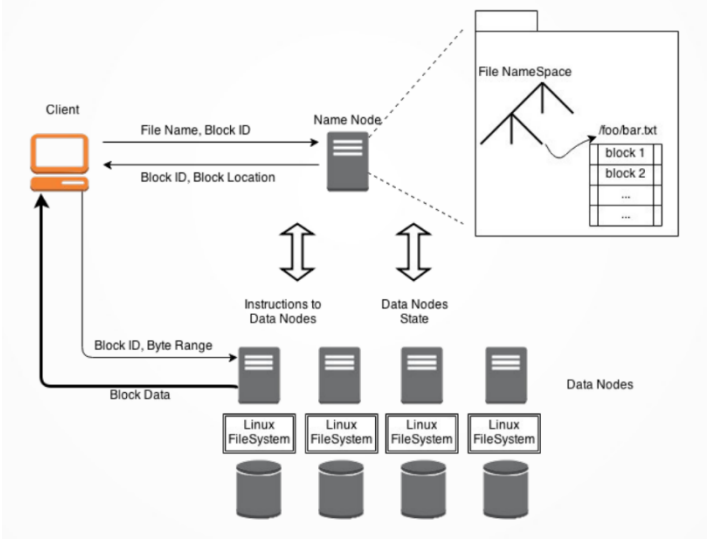
- **Large datasets, that can't be stored on a single machine**, so each "file" is partitioned in several machines. Since it is network-based, we need to consider all the complications related to it, but it is failure-tolerant.
- **One distributed filesystem design**, tailored to read-intensive workloads and throughput, not latency (sequential reads). It is made for commodity hardware.

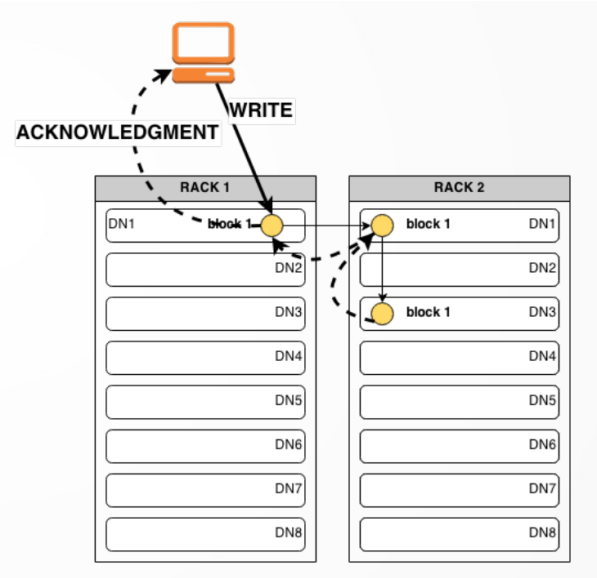
Big files are broken in chunks called **blocks**. The default dimension for each block is 128MB, which is considerably large to make seek times small when compared to read and to ease handling metadata (see NameNode). The blocks are stored as files on the native filesystem and each block is **replicated** on different machines (to avoid losing some block in case of failure).

There are three types of HDFS **nodes**:

- **NameNode**: keeps metadata in RAM in a directory tree, and index of blocks per file. Its load is kept manageable exactly because there aren't that many blocks thanks to their big size. Writes are written in an atomic and synchronous way on a journal, that is put somewhere on the network to keep the register of the writes in case of failure.
- **Secondary NameNode:** It receives copies of the edit log from the NameNode. When the primary is down, the system uses it and stays read-only. If the journal is on the network, we can switch the secondary to primary.
- **DataNode:** store data, sending a heartbeat to the NameNode with the list of their blocks.

In the scheme below, the client requests a file, and the NameNode returns the DataNodes containing the blocks of the file, in a list from the closest to the farthest (since we give preference to closest nodes). Then the client request to the blocks to the corresponding data nodes. In the case of writing in a block, there is a **pipeline replication** in which the first replica is off-rack and second replica is in the same rack of the first, to have a tradeoff between reliability and cluster bandwidth. The default is 3 blocks (the original and two replicas), ant the client receives an acknowledgment after the writes are done.





A **MapReduce job** runs on a set of blocks specified by the programmer. It has one phase each of map, shuffle & reduce. Map and reduce phases are **divided in tasks**, which are **single-machine, independent job** (the only interconnected part is in the shuffle phase). Each machine runs a configurable number of tasks (often one task per CPU, so they don't slow each other down).

- **Map:** by default, 1 HDFS block (1 task). The scheduler will do whatever is possible to run the tasks on a machine having that block. Map tasks are usually quick.
- **Reduce:** Number of reduce tasks is user-specified. Keys are partitioned randomly based on hash values, so one task will handle several keys, but users can override this and write a **custom partitioner** (useful to handle skewed data). Reduce tasks have very variable runtime.

The types of **schedulers** for the jobs are:

- **FIFO:** As soon as a machine is free, it's given the first pending task by the first job. It penalizes small jobs which can wait forever when very large jobs are there.
- **Fair:** Give precedence to active jobs with least running tasks, which results in each job having roughly the same amount of work done in a given moment. Can be configured to kill running tasks to free up space for jobs and you can't prioritize jobs in a queue.

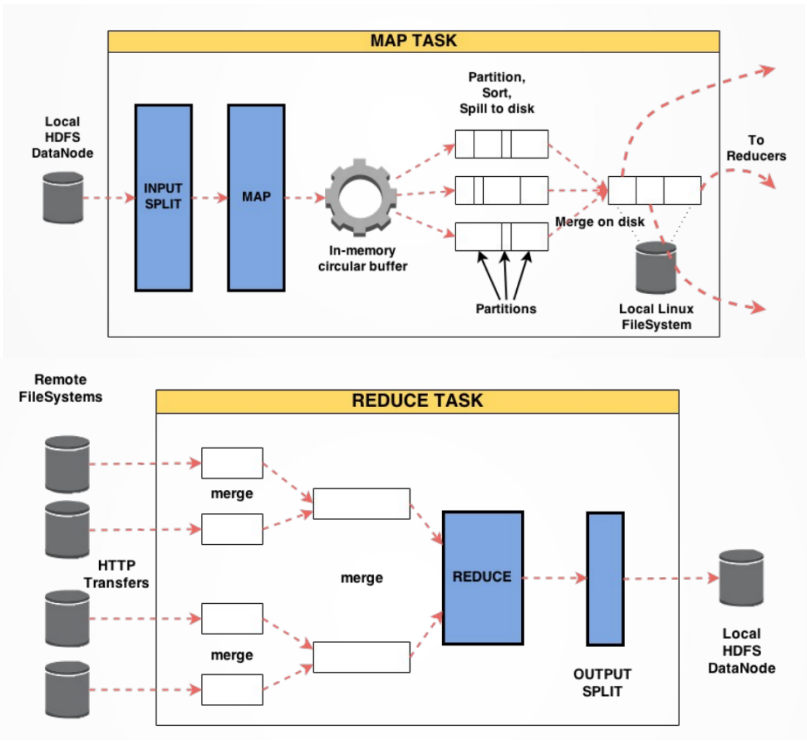
- **Capacity:** Creates "virtual clusters" with a queue each and a dedicated amount of resources. Can be used to make sure that organizations/application have access to a reasonable amount of computing power and if a queue leaves unused resources, they can be used by another queue.

Shortest Job First is not considered here because you generally **don't know** how long a job will need to run.

Considering **failures** instead:

- If a task fails, it's retried a few times. After that, by default the job is marked as failed.
- If a task hangs (no progress), it's killed and retried.
- If a worker machine fails, the scheduler notices the lack of heartbeats and removes it from the worker pool.
- If the scheduler fails, Zookeeper can be used to set up a backup and keep it updated.

Going in depth on the shuffle (and sort) phase:



On the map side, **the output of map stays in a buffer in memory**. When the buffer is filled, it's partitioned (by destination reducer), sorted and saved to disk (reduce keys are always sorted in Hadoop). At the end of a map phase, spills are merged and sent to reducers. Combiners are run right before spilling to disk to save time.

On the reduce side, reducers **fetch** data from mappers and run a merge. Mappers don't delete data right after it's sent to reducers in the case that we need to retry the task after a failure. Output is saved (and replicated) on HDFS.

Part B

- **Apache Spark**

MapReduce has been a big improvement for "big data" on **large clusters of unreliable machines**. However, it's less than perfect for important use cases (Multi-stage applications and interactive ad-hoc queries), because **each MapReduce job has to read and write from disk** and replicate (because it is the solution to unreliability), but this is slow. The goal of Spark is to keep the results in RAM instead and maintain **fault tolerance** at the same time.

- **Spark Internals:**