

Apr 28 · 25 min read

# A Thorough Introduction to Distributed Systems

What is a Distributed System and why is it so complicated?

A bear contemplating distributed systems

## Table of Contents

. . .

# Introduction

With the ever-growing technological expansion of the world, distributed systems are becoming more and more widespread. They are a vast and complex field of study in computer science.

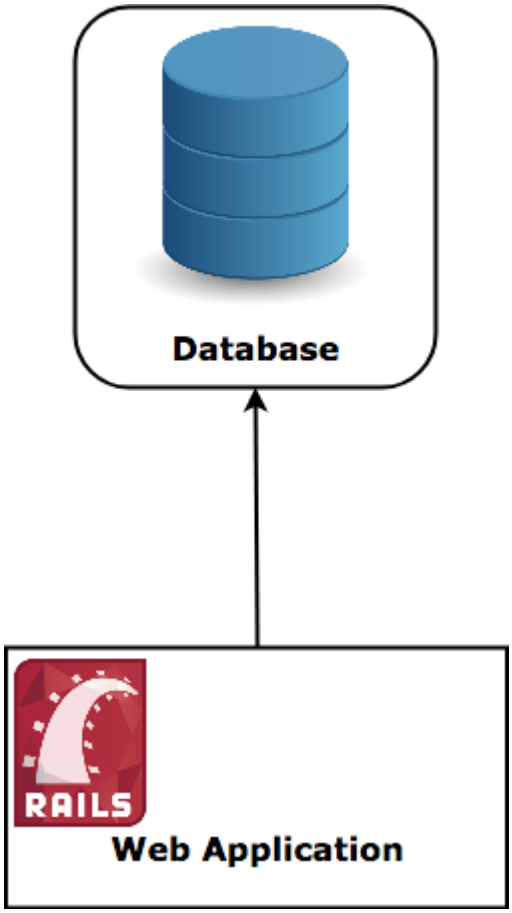
This article aims to introduce you to distributed systems in a basic manner, showing you a glimpse of the different categories of such systems while not diving deep into the details.

## What is a distributed system?

A distributed system in its most simplest definition is a group of computers working together as to appear as a single computer to the end-user.

These machines have a shared state, operate concurrently and can fail independently without affecting the whole system's uptime.

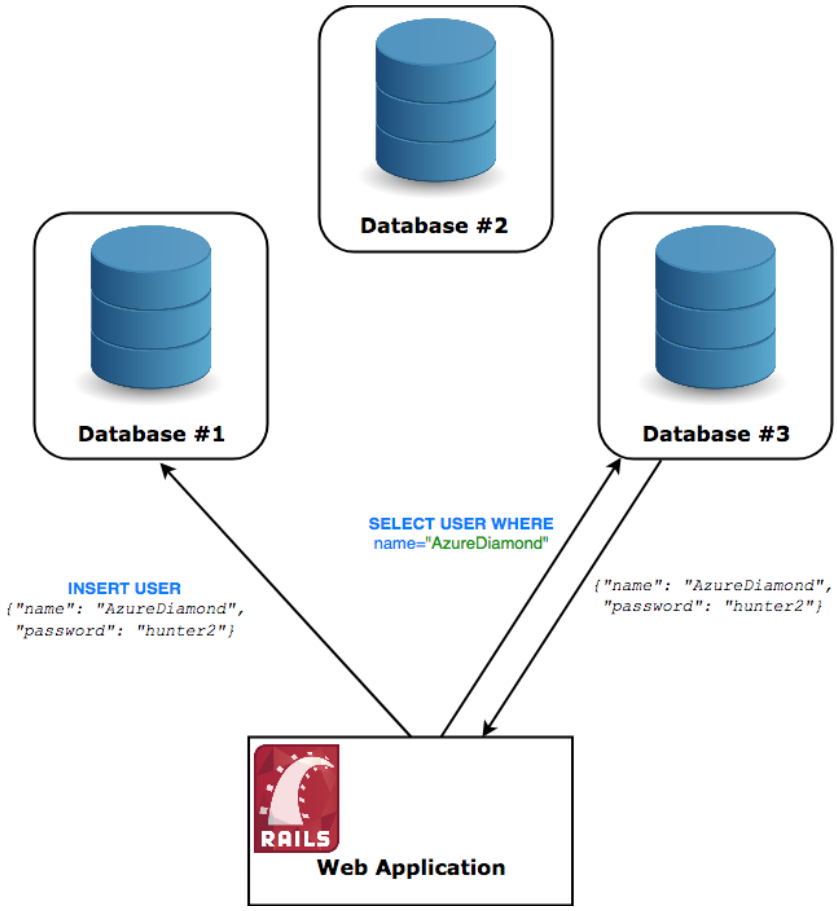
I propose we incrementally work through an example of distributing a system so that you can get a better sense of it all:




A traditional stack

Let's go with a database! Traditional databases are stored on the filesystem of one single machine, whenever you want to fetch/insert information in it—you talk to that machine directly.

For us to distribute this database system, we'd need to have this database run on multiple machines at the same time. The user must be able to talk to whichever machine he chooses and should not be able to tell that he is not talking to a single machine—if he inserts a record into node#1, node #3 must be able to return that record.




An architecture that can be considered distributed

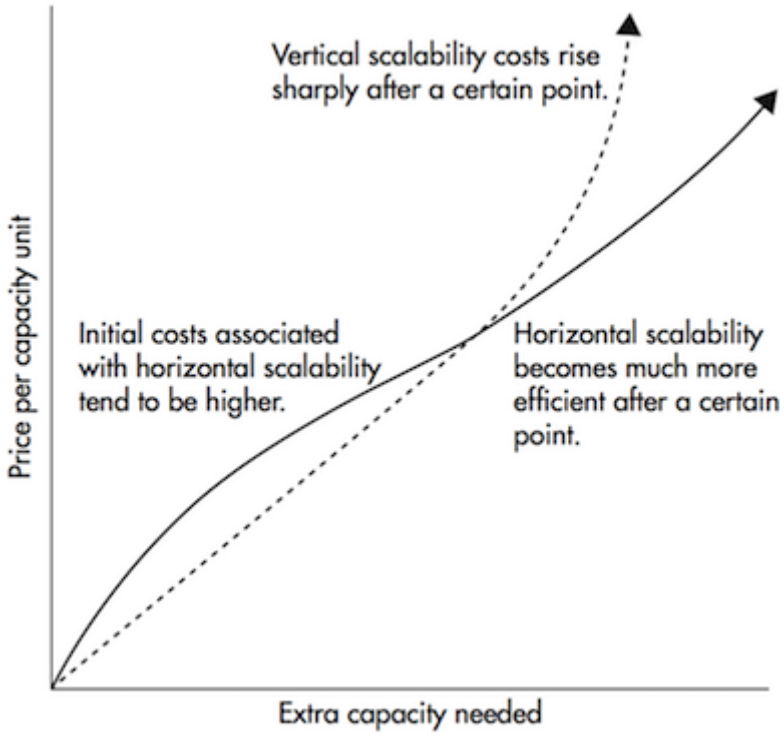
## Why distribute a system?

Systems are always distributed by necessity. The truth of the matter is —managing distributed systems is a complex topic chock-full of pitfalls and landmines. It is a headache to deploy, maintain and debug distributed systems, so why go there at all?

What a distributed system enables you to do is **scale horizontally**. Going back to our previous example of the single database server, the only way to handle more traffic would be to upgrade the hardware the database is running on. This is called **scaling vertically**.

Scaling vertically is all well and good while you can, but after a certain point you will see that even the best hardware is not sufficient for enough traffic, not to mention impractical to host.

**Scaling horizontally** simply means adding more computers rather than upgrading the hardware of a single one.




Horizontal scaling becomes **much cheaper** after a certain threshold

It is significantly cheaper than vertical scaling after a certain threshold but that is not its main case for preference.

Vertical scaling can only bump your performance up to the latest hardware's capabilities. These capabilities prove to be **insufficient** for technological companies with moderate to big workloads.

The best thing about horizontal scaling is that you have no cap on how much you can scale—whenever performance degrades you simply add another machine, up to infinity potentially.

Easy scaling is not the only benefit you get from distributed systems. ***Fault tolerance*** and ***low latency*** are also equally as important.

***Fault Tolerance***—a cluster of ten machines across two data centers is inherently more fault-tolerant than a single machine. Even if one data center catches on fire, your application would still work.

***Low Latency***—The time for a network packet to travel the world is physically bounded by the speed of light. For example, the shortest

possible time for a request's **round-trip time** (that is, go back and forth) in a fiber-optic cable between New York to Sydney is 160ms. Distributed systems allow you to have a node in both cities, allowing traffic to hit the node that is closest to it.

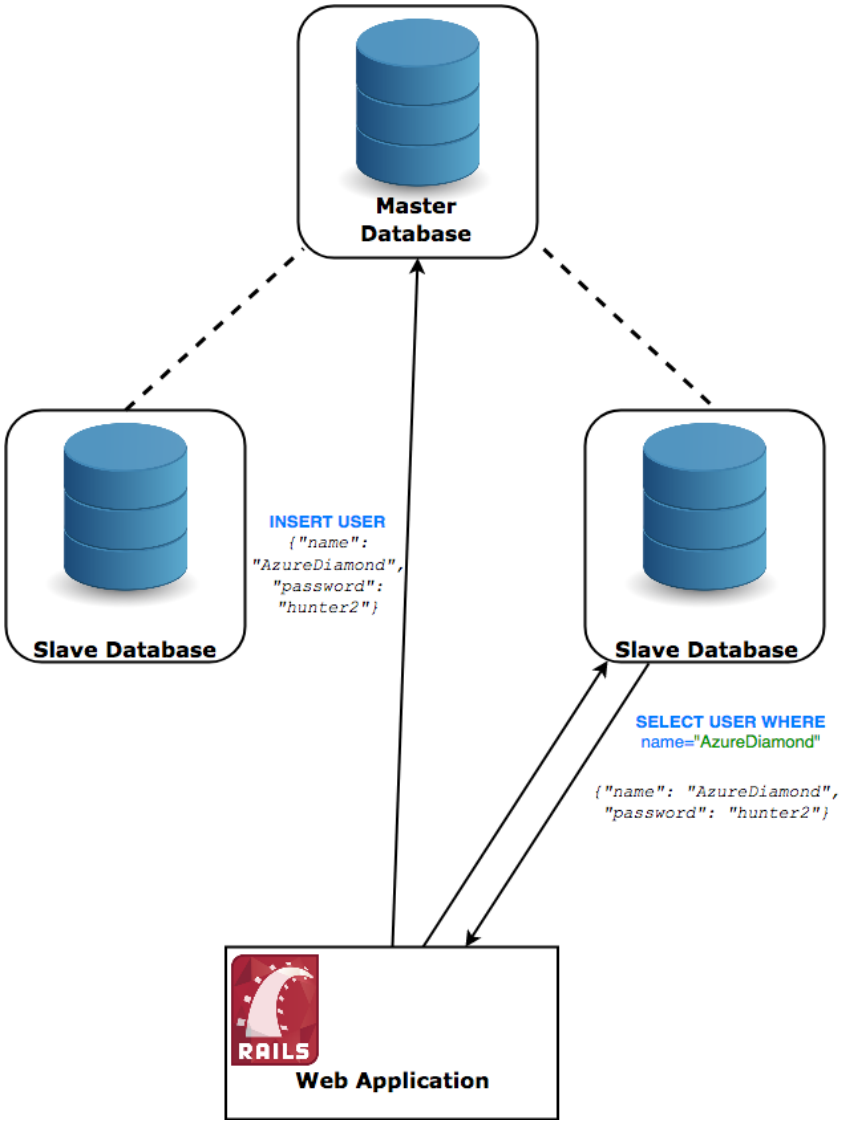
For a distributed system to work, though, you need the software running on those machines to be specifically designed for running on multiple computers at the same time and handling the problems that come along with it. This turns out to be no easy feat.

## Scaling our database

Imagine that our web application got insanely popular. Imagine also that our database started getting twice as much queries per second as it can handle. Your application would immediately start to decline in performance and this would get noticed by your users.

Let's work together and make our database scale to meet our high demands.

In a typical web application you normally read information much more frequently than you insert new information or modify old one.




There is a way to increase read performance and that is by the so-called **Master-Slave Replication** strategy. Here, you create two new database servers which sync up with the main one. The catch is that you can **only read** from these new instances.

Whenever you insert or modify information—you talk to the master database. It, in turn, asynchronously informs the slaves of the change and they save it as well.

Congratulations, you can now execute 3x as much read queries! Isn't this great?

## Pitfall

**Gotcha!** We immediately lost the ***C*** in our relational database's **ACID** guarantees, which stands for Consistency.

You see, there now exists a possibility in which we insert a new record into the database, immediately afterwards issue a read query for it and get nothing back, as if it didn't exist!

Propagating the new information from the master to the slave does not happen instantaneously. There actually exists a time window in which you can fetch stale information. If this were not the case, your write performance would suffer, as it would have to synchronously wait for the data to be propagated.

Distributed systems come with a handful of trade-offs. This particular issue is one you will have to live with if you want to adequately scale.

**Continuing to Scale**

Using the slave database approach, we can horizontally scale our read traffic up to some extent. That's great but we've hit a wall in regards to our write traffic—it's still all in one server!

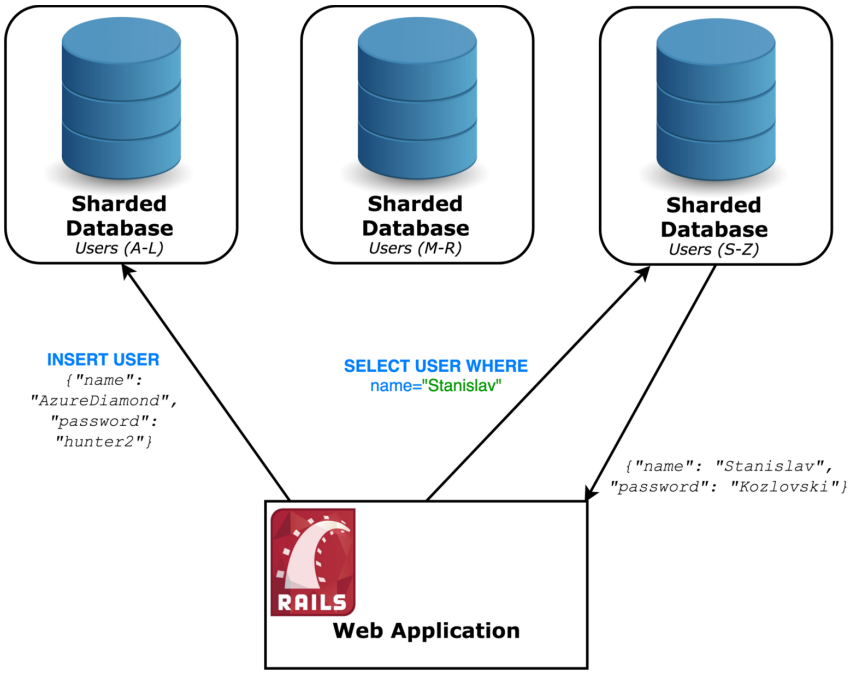
We're not left with much options here. We simply need to split our write traffic into multiple servers as one is not able to handle it.

One way is to go with a multi-master replication strategy. There, instead of slaves that you can only read from, you have multiple master nodes which support reads and writes. Unfortunately, this gets complicated real quick as you now have the ability to create conflicts (e.g insert two records with same ID).

Let's go with another technique called ***sharding*** (also called ***partitioning***).

With sharding you split your server into multiple smaller servers, called ***shards.*** These shards all hold different records—you create a rule as to what kind of records go into which shard. It is very important to create the rule such that the data gets spread in an **uniform way**.

A possible approach to this is to define ranges according to some information about a record (e.g users with name A-D).




This sharding key should be chosen very carefully, as the load is not always equal based on arbitrary columns. (e.g more people have a name starting with *C* rather than *Z*). A single shard that receives more requests than others is called a ***hot spot*** and must be avoided. Once split up, re-sharding data becomes incredibly expensive and can cause significant downtime, as was the case with FourSquare's infamous 11 hour outage.

To keep our example simple, assume our client (the Rails app) knows which database to use for each record. It is also worth noting that there are many strategies for sharding and this is a simple example to illustrate the concept.

We have won quite a lot right now—we can increase our write traffic ***N*** times where ***N*** is the number of shards. This practically gives us almost no limit—imagine how finely-grained we can get with this partitioning.

## Pitfall

Everything in Software Engineering is more or less a trade-off and this is no exception. Sharding is no simple feat and is best avoided until really needed.

We have now made queries by keys **other than the partitioned key** incredibly inefficient (they need to go through all of the shards). SQL

`JOIN` queries are even worse and complex ones become practically unusable.

### Decentralized vs Distributed

Before we go any further I'd like to make a distinction between the two terms.

Even though the words sound similar and can be concluded to mean the same logically, their difference makes a significant technological and political impact.

***Decentralized*** is still ***distributed*** in the technical sense, but the whole decentralized systems is not owned by one actor. No one company can own a decentralized system, otherwise it wouldn't be decentralized anymore.

This means that most systems we will go over today can be thought of as ***distributed centralized systems***—and that is what they're made to be.

If you think about it—it is harder to create a decentralized system because then you need to handle the case where some of the participants are malicious. This is not the case with normal distributed systems, as you know you own all the nodes.

*Note: This definition has been debated a lot and can be confused with others (peer-to-peer, federated). In early literature, it's been defined differently as well. Regardless, what I gave you as a definition is what I feel is the most widely used now that blockchain and cryptocurrencies popularized the term.*

. . .

## Distributed System Categories

We are now going to go through a couple of distributed system categories and list their largest publicly-known production usage. Bear in mind that most such numbers shown are outdated and are most probably significantly bigger as of the time you are reading this.

# Distributed Data Stores

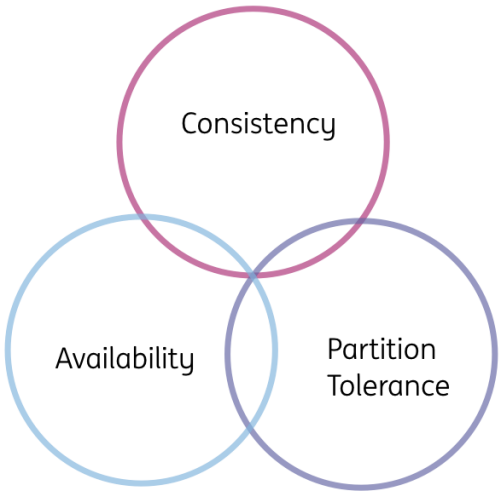
Distributed Data Stores are most widely used and recognized as Distributed Databases. Most distributed databases are NoSQL non-relational databases, limited to key-value semantics. They provide incredible performance and scalability at the cost of consistency or availability.

> ***Known Scale***—*Apple is known to use 75,000 Apache Cassandra nodes storing over 10 petabytes of data, back in 2015*

We cannot go into discussions of distributed data stores without first introducing the **CAP Theorem.**

## CAP Theorem

Proven way back in 2002, the CAP theorem states that a distributed data store cannot simultaneously be consistent, available and partition tolerant.




Choose 2 out of 3 (But not Consistency and Availability)

Some quick definitions:

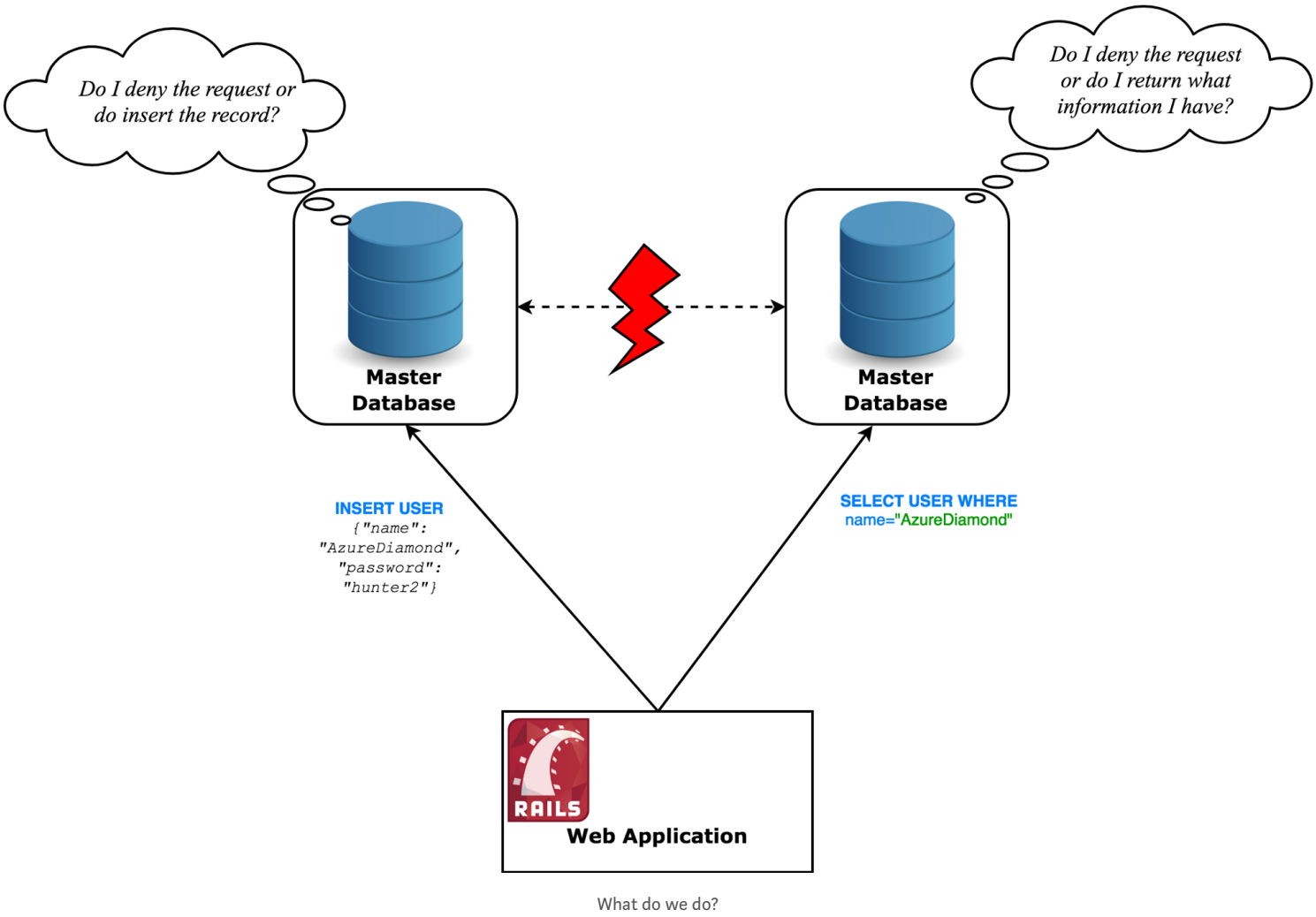
- **Consistency**—What you read and write sequentially is what is expected (remember the gotcha with the database replication a

few paragraphs ago?)

- **Availability**—the whole system does not die—every non-failing node always returns a response.
- **Partition Tolerant**—The system continues to function and uphold its consistency/availability guarantees in spite of network partitions

In reality, partition tolerance must be a given for any distributed data store. As mentioned in many places, one of which this great article, you cannot have consistency and availability without partition tolerance.

Think about it: if you have two nodes which accept information and their connection dies—how are they both going to be available and simultaneously provide you with consistency? They have no way of knowing what the other node is doing and as such have can either become offline *(unavailable)* or work with stale information *(inconsistent)*.




What do we do?

In the end you're left to choose if you want your system to be strongly consistent or highly available ***under a network partition***.

Practice shows that most applications value availability more. You do not necessarily always need strong consistency. Even then, that trade-off is not necessarily made because you need the 100% availability guarantee, but rather because network latency can be an issue when having to synchronize machines to achieve strong consistency. These and more factors make applications typically opt for solutions which offer high availability.

Such databases settle with the weakest consistency model—***eventual consistency*** *(strong vs eventual consistency explanation)*. This model guarantees that if no new updates are made to a given item, **eventually** all accesses to that item will return the latest updated value.

Those systems provide **BASE** properties (as opposed to traditional databases' ACID)

- **B**asically **A**vailable—The system always returns a response
- **S**oft state—The system could change over time, even during times of no input (due to eventual consistency)
- **E**ventual consistency—In the absence of input, the data will spread to every node sooner or later—thus becoming consistent

Examples of such available distributed databases—Cassandra, Riak, Voldemort

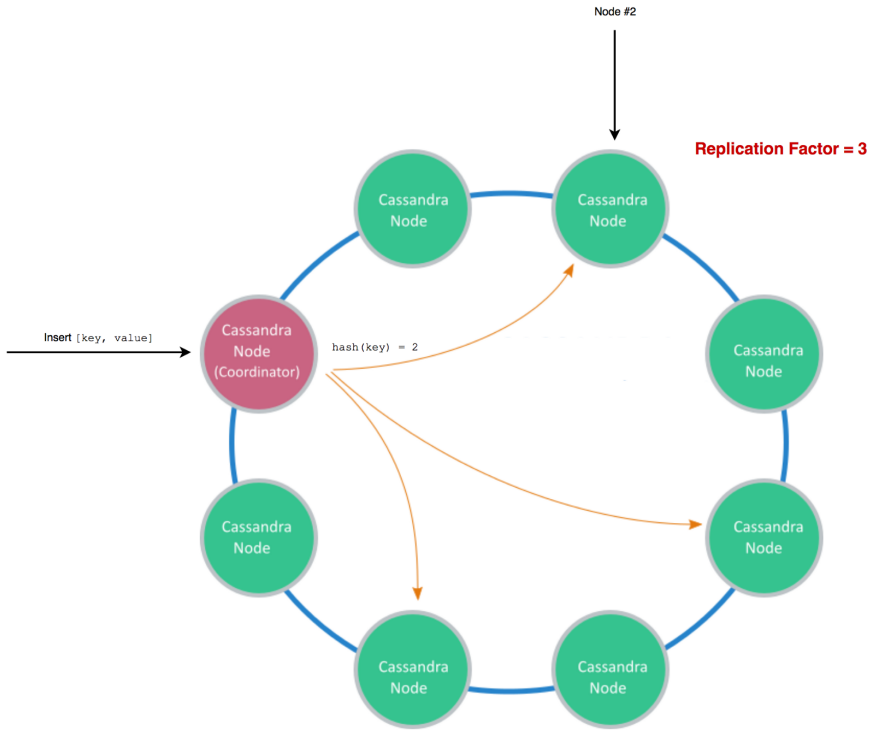
Of course, there are other data stores which prefer stronger consistency —HBase, Couchbase, Redis, Zookeeper

The CAP theorem is worthy of multiple articles on its own—some regarding how you can tweak a system's CAP properties depending on how the client behaves and others on how it is not understood properly.

## Cassandra

Cassandra, as mentioned above, is a distributed No-SQL database which prefers the AP properties out of the CAP, settling with eventual consistency. I must admit this may be a bit misleading, as Cassandra is highly configurable—you can make it provide strong consistency at the expense of availability as well, but that is not its common use case.

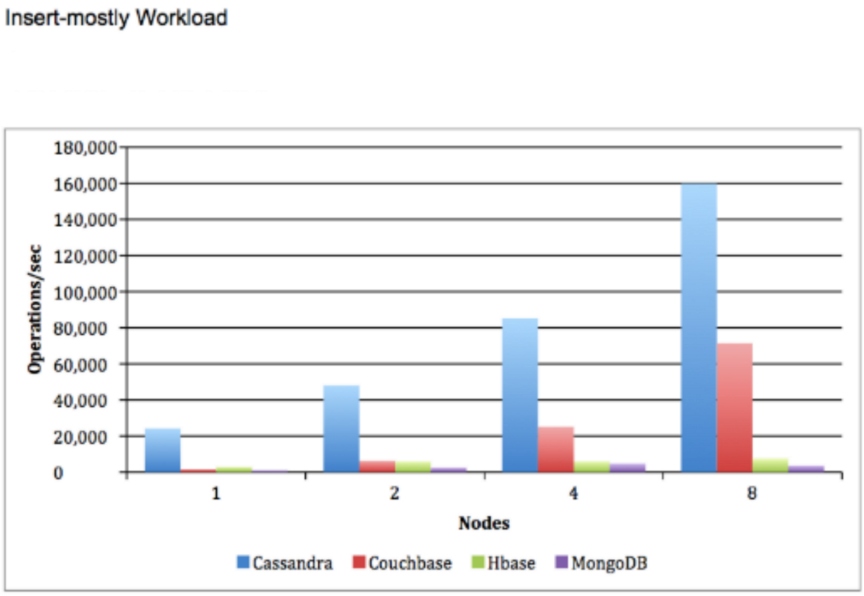
Cassandra uses consistent hashing to determine which nodes out of your cluster must manage the data you are passing in. You set a **replication factor**, which basically states to how many nodes you want to replicate your data.




Sample write

When reading, you will read from those nodes only.

Cassandra is massively scalable, providing absurdly high write throughput.




| Nodes | Cassandra | Couchbase | HBase | MongoDB |
|---|---|---|---|---|
| 1 | ~24,000 | ~1,600 | ~2,900 | ~1,000 |
| 2 | ~48,000 | ~6,200 | ~5,800 | ~2,400 |
| 4 | ~85,300 | ~25,200 | ~5,900 | ~4,800 |
| 8 | ~160,000 | ~71,300 | ~7,600 | ~3,600 |

Possibly biased diagram, showing writes per second benchmarks. Taken from here.

Even though this diagram might be biased and it looks like it compares Cassandra to databases set to provide strong consistency (otherwise I can't see why MongoDB would drop performance when upgraded from 4 to 8 nodes), this should still show what a properly set up Cassandra cluster is capable of.

Regardless, in the distributed systems trade-off which enables horizontal scaling and incredibly high throughput, Cassandra does not provide some fundamental features of ACID databases—namely, transactions.

## Consensus

Database transactions are tricky to implement in distributed systems as they require each node to agree on the right action to take (abort or commit). This is known as ***consensus*** and it is a fundamental problem in distributed systems.

Reaching the type of agreement needed for the "transaction commit" problem is straightforward if the participating processes and the network are completely reliable. However, real systems are subject to a

number of possible faults, such as process crashes, network partitioning, and lost, distorted, or duplicated messages.

This poses an issue—it has been proven impossible to guarantee that a correct consensus is reached within a bounded time frame on a non-reliable network.

In practice, though, there are algorithms that reach consensus on a non-reliable network pretty quickly. Cassandra actually provides lightweight transactions through the use of the Paxos algorithm for distributed consensus.

. . .

## Distributed Computing

Distributed computing is the key to the influx of Big Data processing we've seen in recent years. It is the technique of splitting an enormous task (e.g aggregate 100 billion records), of which no single computer is capable of practically executing on its own, into many smaller tasks, each of which can fit into a single commodity machine. You split your huge task into many smaller ones, have them execute on many machines in parallel, aggregate the data appropriately and you have solved your initial problem. This approach again enables you to scale horizontally—when you have a bigger task, simply include more nodes in the calculation.

> *Known Scale—Folding@Home had 160k active machines in 2012*

An early innovator in this space was Google, which by necessity of their large amounts of data had to invent a new paradigm for distributed computation—MapReduce. They published a paper on it in 2004 and the open source community later created Apache Hadoop based on it.

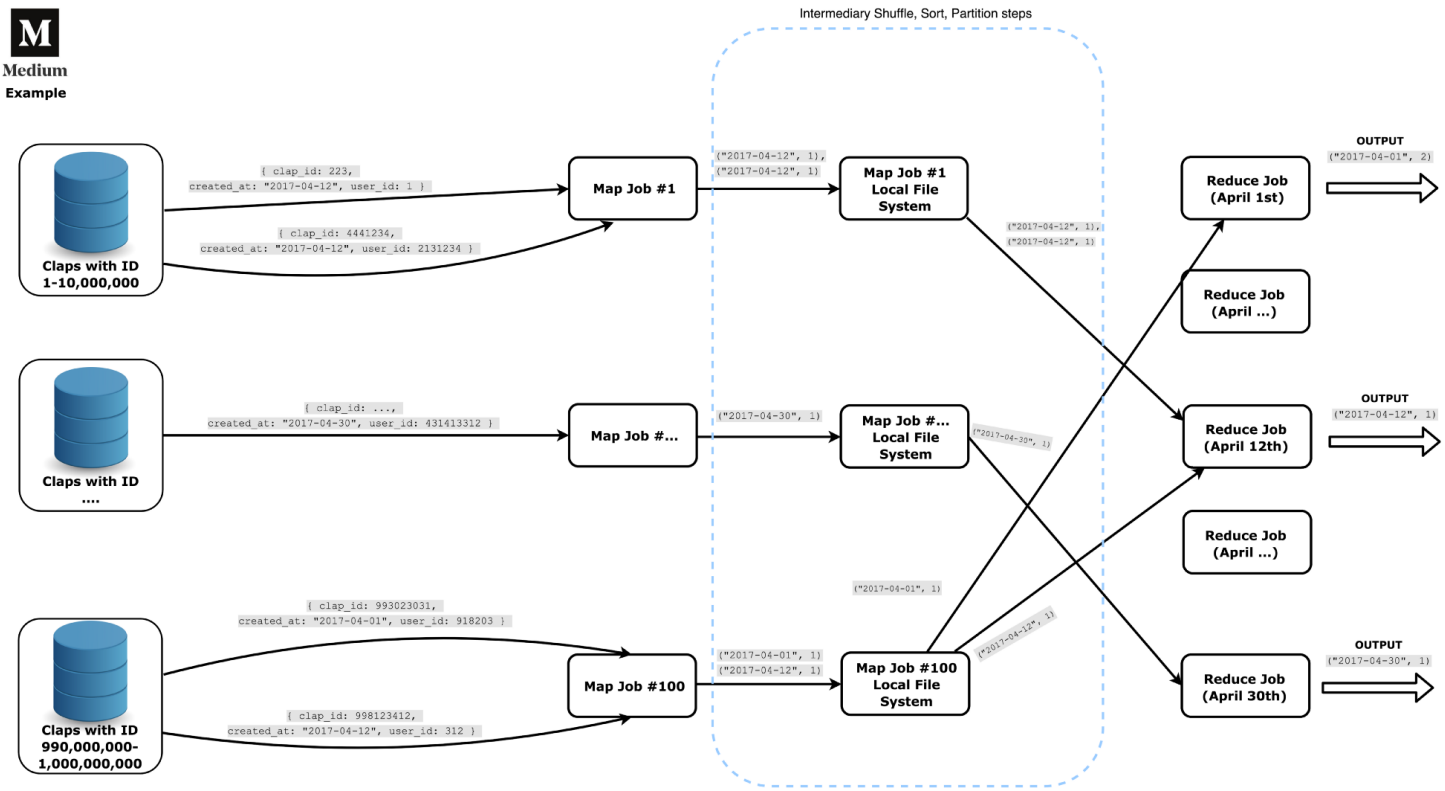
### MapReduce

MapReduce can be simply defined as two steps—mapping the data and reducing it to something meaningful.

Let's get at it with an example again:

Say we are Medium and we stored our enormous information in a secondary distributed database for warehousing purposes. We want to fetch data representing the number of claps issued each day throughout April 2017 (a year ago).

This example is kept as short, clear and simple as possible, but imagine we are working with loads of data (e.g analyzing billions of claps). We won't be storing all of this information on one machine obviously and we won't be analyzing all of this with one machine only. We also won't be querying the production database but rather some "warehouse" database built specifically for low-priority offline jobs.




Each Map job is a separate node transforming as much data as it can. Each job traverses all of the data in the given storage node and maps it to a simple tuple of the date and the number one. Then, three intermediary steps *(which nobody talks about)* are done—Shuffle, Sort and Partition. They basically further arrange the data and delete it to the appropriate reduce job. As we're dealing with big data, we have each Reduce job separated to work on a single date only.

This is a good paradigm and surprisingly enables you to do a lot with it—you can chain multiple MapReduce jobs for example.

### Better Techniques

MapReduce is somewhat legacy nowadays and brings some problems with it. Because it works in batches (jobs) a problem arises where if your job fails—you need to restart the whole thing. A 2-hour job failing can really slow down your whole data processing pipeline and you do not want that in the very least, especially in peak hours.

Another issue is the time you wait until you receive results. In real-time analytic systems (which all have big data and thus use distributed computing) it is important to have your latest crunched data be as fresh as possible and certainly not from a few hours ago.

As such, other architectures have emerged that address these issues. Namely Lambda Architecture (mix of batch processing and stream processing) and Kappa Architecture (only stream processing). These advances in the field have brought new tools enabling them—Kafka Streams, Apache Spark, Apache Storm, Apache Samza.

. . .

## Distributed File Systems

Distributed file systems can be thought of as distributed data stores. They're the same thing as a concept—storing and accessing a large amount of data across a cluster of machines all appearing as one. They typically go hand in hand with Distributed Computing.

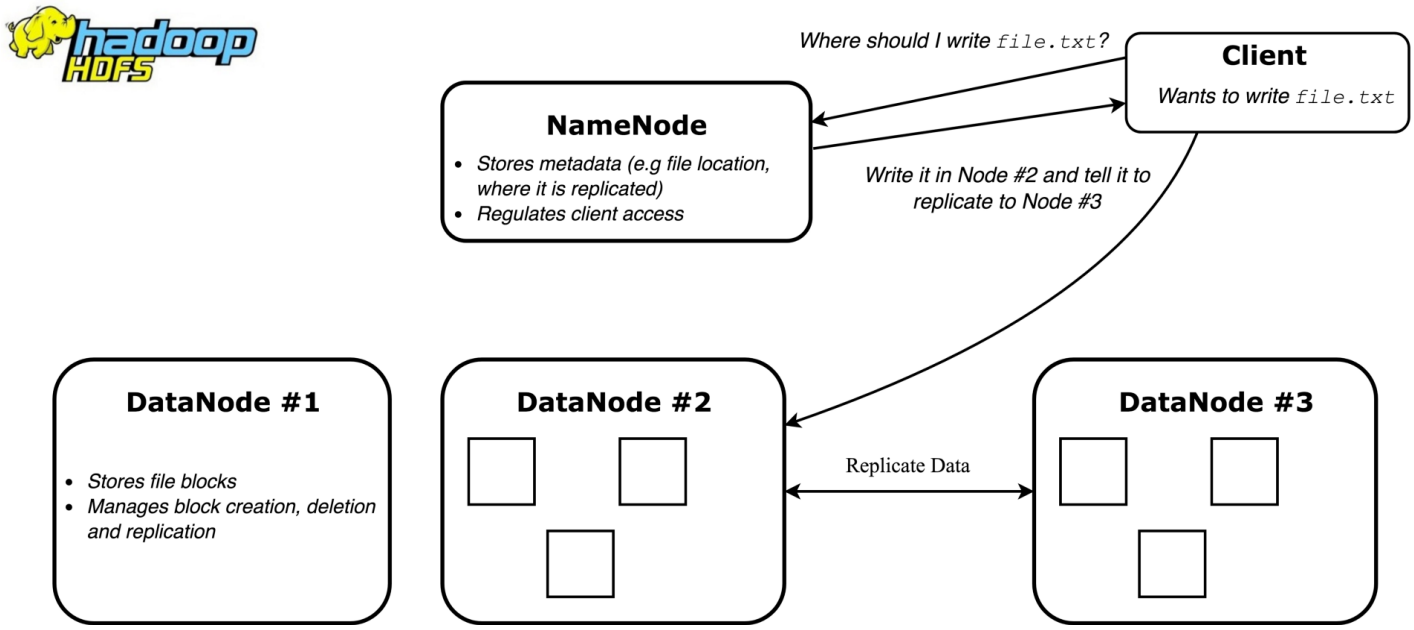
> *Known Scale—Yahoo is known for running HDFS on over 42,000 nodes for storage of 600 Petabytes of data, way back in 2011*

Wikipedia defines the difference being that distributed file systems allow files to be accessed using the same interfaces and semantics as local files, not through a custom API like the Cassandra Query Language (CQL).

### HDFS

Hadoop Distributed File System (HDFS) is the distributed file system used for distributed computing via the Hadoop framework. Boasting widespread adoption, it is used to store and replicate large files (GB or TB in size) across many machines.

Its architecture consists mainly of ***NameNodes*** and ***DataNodes***. NameNodes are responsible for keeping metadata about the cluster, like which node contains which file blocks. They act as coordinators for the network by figuring out where best to store and replicate files, tracking the system's health. DataNodes simply store files and execute commands like replicating a file, writing a new one and others.




Unsurprisingly, HDFS is best used with Hadoop for computation as it provides data awareness to the computation jobs. Said jobs then get ran on the nodes storing the data. This leverages data locality—optimizes computations and reduces the amount of traffic over the network.

## IPFS

Interplanetary File System (IPFS) is an exciting new peer-to-peer protocol/network for a distributed file system. Leveraging Blockchain technology, it boasts a completely decentralized architecture with no single owner nor point of failure.

IPFS offers a naming system (similar to DNS) called IPNS and lets users easily access information. It stores file via historic versioning, similar to how Git does. This allows for accessing all of a file's previous states.

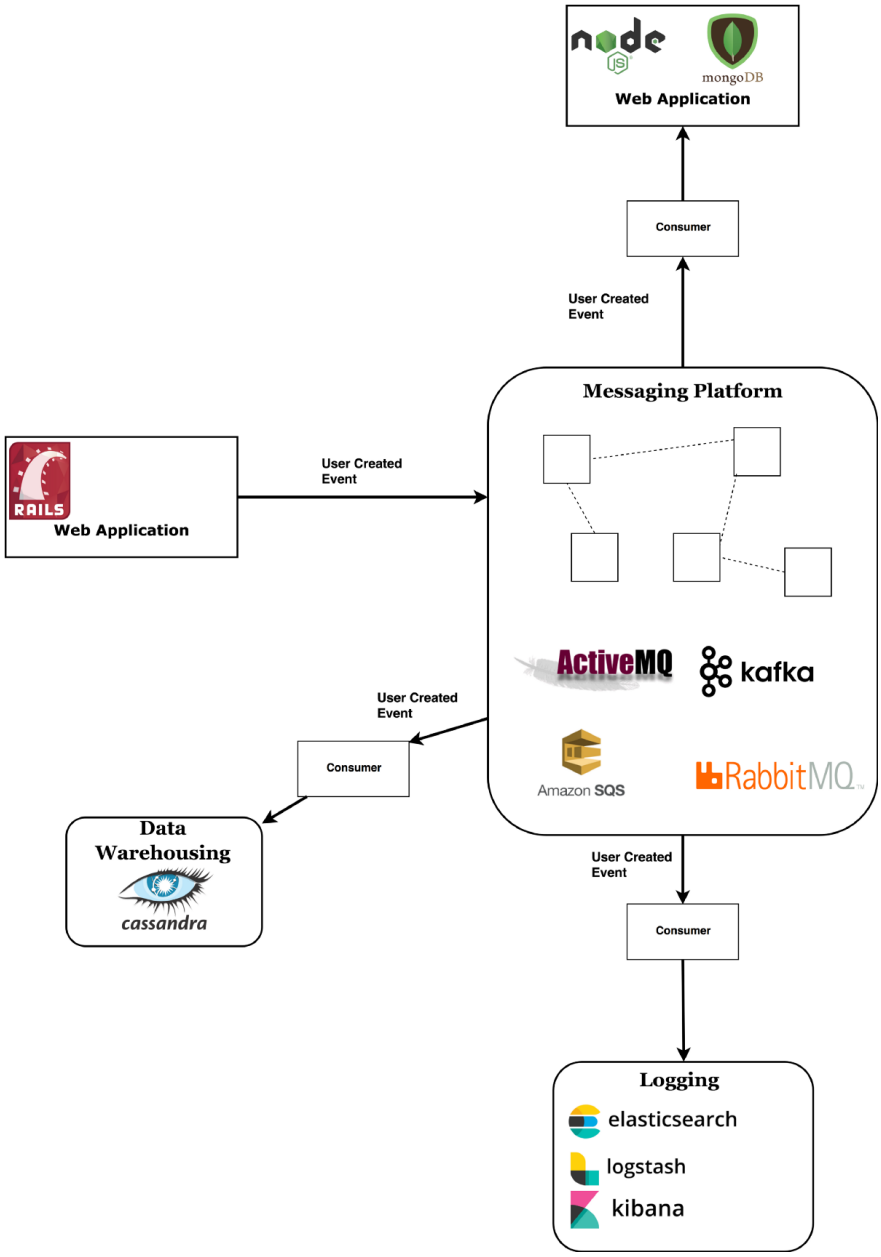
It is still undergoing heavy development (v0.4 as of time of writing) but has already seen projects interested in building over it (FileCoin).

. . .

## Distributed Messaging

Messaging systems provide a central place for storage and propagation of messages/events inside your overall system. They allow you to decouple your application logic from directly talking with your other systems.

> *Known Scale—LinkedIn's Kafka cluster processed 1 trillion messages a day with peaks of 4.5 millions messages a second.*




Simply put, a messaging platform works in the following way:

A message is broadcast from the application which potentially create it (called a **producer**), goes into the platform and is read by potentially multiple applications which are interested in it (called **consumer**s).

If you need to save a certain event to a few places (e.g user creation to database, warehouse, email sending service and whatever else you can come up with) a messaging platform is the cleanest way to spread that message.

Consumers can either pull information out of the brokers (pull model) or have the brokers push information directly into the consumers (push model).

There are a couple of popular top-notch messaging platforms:

RabbitMQ—Message broker which allows you finer-grained control of message trajectories via routing rules and other easily configurable settings. Can be called a smart broker, as it has a lot of logic in it and tightly keeps track of messages that pass through it. Provides settings for both **AP** and **CP** from **CAP**. Uses a push model for notifying the consumers.

Kafka—Message broker (and all out platform) which is a bit lower level, as in it does not keep track of which messages have been read and does not allow for complex routing logic. This helps it achieve amazing performance. In my opinion, this is the biggest prospect in this space with active development from the open-source community and support from the Confluent team. Kafka arguably has the most widespread use from top tech companies. I wrote a thorough introduction to this, where I go into detail about all of its goodness.

Apache ActiveMQ—The oldest of the bunch, dating from 2004. Uses the JMS API, meaning it is geared towards Java EE applications. It got rewritten as ActiveMQ Artemis, which provides outstanding performance on par with Kafka.

Amazon SQS—A messaging service provided by AWS. Lets you quickly integrate it with existing applications and eliminates the need to handle your own infrastructure, which might be a big benefit, as systems like Kafka are notoriously tricky to set up. Amazon also offers two similar services—SNS and MQ, the latter of which is basically ActiveMQ but managed by Amazon.

## Distributed Applications

If you roll up 5 Rails servers behind a single load balancer all connected to one database, could you call that a distributed application? Recall my definition from up above:

> *A distributed system is a group of computers working together as to appear as a single computer to the end-user. These machines have a shared state,*

> *operate concurrently and can fail independently without affecting the whole system's uptime.*

If you count the database as a shared state, you could argue that this can be classified as a distributed system—but you'd be wrong, as you've missed the "*working together*" part of the definition.

A system is distributed only if the nodes communicate with each other to coordinate their actions.

Therefore something like an application running its back-end code on a peer-to-peer network can better be classified as a distributed application. Regardless, this is all needless classification that serves no purpose but illustrate how fussy we are about grouping things together.

> *Known Scale—BitTorrent swarm of 193,000 nodes for an episode of Game of Thrones, April, 2014*

## Erlang Virtual Machine

Erlang is a functional language that has great semantics for concurrency, distribution and fault-tolerance. The Erlang Virtual Machine itself handles the distribution of an Erlang application.

Its model works by having many **isolated** lightweight processes all with the ability to talk to each other via a built-in system of message passing. This is called the ***Actor Model*** and the Erlang OTP libraries can be thought of as a distributed actor framework (along the lines of Akka for the JVM).

The model is what helps it achieve great concurrency rather simply—the processes are spread across the available cores of the system running them. Since this is indistinguishable from a network setting (apart from the ability to drop messages), Erlang's VM can connect to other Erlang VMs running in the same data center or even in another continent. This swarm of virtual machines run one single application and handle machine failures via takeover (another node gets scheduled to run).

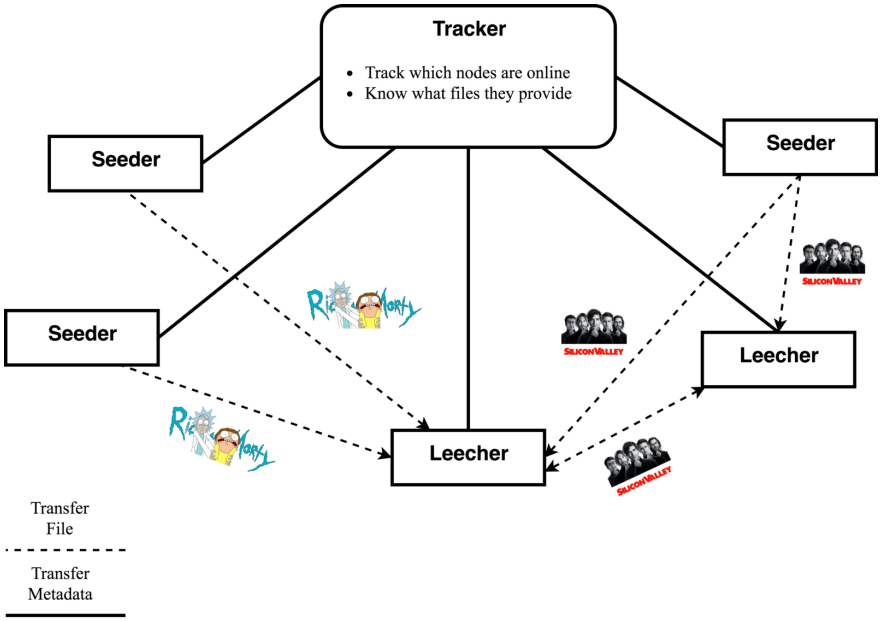
In fact, the distributed layer of the language was added in order to provide fault tolerance. Software running on a single machine is always at risk of having that single machine dying and taking your application

offline. Software running on many nodes allows easier hardware failure handling, provided the application was built with that in mind.

## BitTorrent

BitTorrent is one of the most widely used protocol for transferring large files across the web via torrents. The main idea is to facilitate file transfer between different peers in the network without having to go through a main server.

Using a BitTorrent client, you connect to multiple computers across the world to download a file. When you open a .torrent file, you connect to a so-called ***tracker***, which is a machine that acts as a coordinator. It helps with peer discovery, showing you the nodes in the network which have the file you want.




a sample network

You have the notions of two types of user, a ***leecher*** and a ***seeder***. A leecher is the user who is downloading a file and a seeder is the user who is uploading said file.

The funny thing about peer-to-peer networks is that you, as an ordinary user, have the ability to join and contribute to the network.

BitTorrent and its precursors (Gnutella, Napster) allow you to voluntarily host files and upload to other users who want them. The

reason BitTorrent is so popular is that it was the first of its kind to provide incentives for contributing to the network. ***Freeriding***, where a user would only download files, was an issue with the previous file sharing protocols.

BitTorrent solved freeriding to an extent by making seeders upload more to those who provide the best download rates. It works by incentivizing you to upload while downloading a file. Unfortunately, after you're done, nothing is making you stay active in the network. This causes a lack of seeders in the network who have the full file and as the protocol relies heavily on such users, solutions like private trackers came into fruition. Private trackers require you to be a member of a community (often invite-only) in order to participate in the distributed network.

After advancements in the field, trackerless torrents were invented. This was an upgrade to the BitTorrent protocol that did not rely on centralized trackers for gathering metadata and finding peers but instead use new algorithms. One such instance is Kademlia (Mainline DHT), a distributed hash table (DHT) which allows you to find peers through other peers. In effect, each user performs a tracker's duties.

. . .

## Distributed Ledgers

A distributed ledger can be thought of as an immutable, append-only database that is replicated, synchronized and shared across all nodes in the distributed network.

> *Known Scale—Ethereum Network had a peak of 1.3 million transactions a day on January 4th, 2018.*

They leverage the Event Sourcing pattern, allowing you to rebuild the ledger's state at any time in its history.

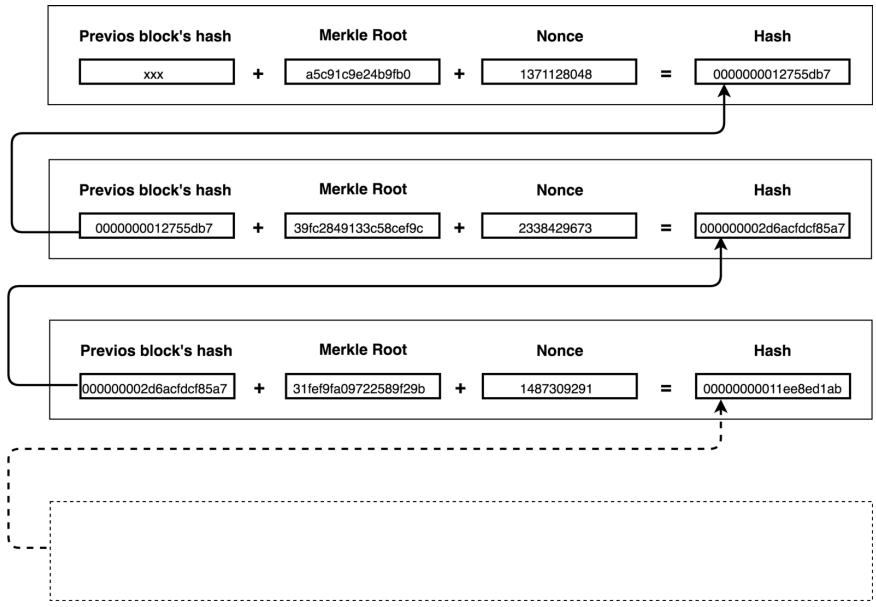
### Blockchain

Blockchain is the current underlying technology used for distributed ledgers and in fact marked their start. This latest and greatest

innovation in the distributed space enabled the creation of the first ever truly distributed payment protocol—Bitcoin.

Blockchain is a distributed ledger carrying an ordered list of all transactions that ever occurred in its network. Transactions are grouped and stored in blocks. The whole blockchain is essentially a linked-list of blocks *(hence the name)*. Said blocks are computationally expensive to create and are tightly linked to each other through cryptography.

Simply said, each block contains a special hash (that starts with X amount of zeroes) of the current block's contents (in the form of a Merkle Tree) plus the previous block's hash. This hash requires a lot of CPU power to be produced because the only way to come up with it is through brute-force.




Simplified blockchain

**Miners** are the nodes who try to compute the hash (via bruteforce). The miners all compete with each other for who can come up with a random string (called a ***nonce***) which, when combine with the contents, produces the aforementioned hash. Once somebody finds the correct nonce—he broadcasts it to the whole network. Said string is then verified by each node on its own and accepted into their chain.

This translates into a system where it is absurdly costly to modify the blockchain and absurdly easy to verify that it is not tampered with.

It is costly to change a block's contents because that would produce a different hash. Remember that each subsequent block's hash is dependent on it. If you were to change a transaction in the first block of the picture above—you would change the Merkle Root. This would in turn change the block's hash (most likely without the needed leading zeroes)—that would change block #2's hash and so on and so on. This means you'd need to brute-force a new nonce for every block after the one you just modified.

The network always trusts and replicates the longest valid chain. In order to cheat the system and **eventually** produce a longer chain you'd need more than 50% of the total CPU power used by all the nodes.

Blockchain can be thought of as a distributed mechanism for ***emergent consensus***. Consensus is not achieved explicitly—there is no election or fixed moment when consensus occurs. Instead, consensus is an ***emergent*** product of the asynchronous interaction of thousands of independent nodes, all following protocol rules.

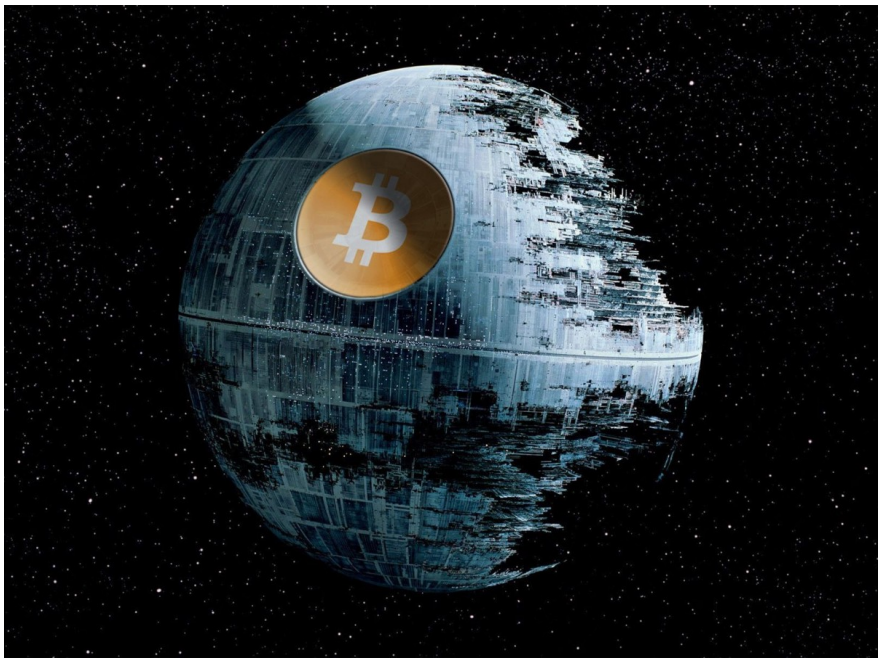
This unprecedented innovation has recently become a boom in the tech space with people predicting it will mark the creation of the Web 3.0. It is definitely the most exciting space in the software engineering world right now, filled with extremely challenging and interesting problems waiting to be solved.

## Bitcoin

What previous distributed payment protocols lacked was a way to practically prevent the double-spending problem in real time, in a distributed manner. Research has produced interesting propositions[1] but Bitcoin was the first to implement a practical solution with clear advantages over others.

The double spending problem states that an actor (e.g Bob) cannot spend his single resource in two places. If Bob has $1, he should not be able to give it to both Alice and Zack—it is only one asset, it cannot be duplicated. It turns out it is really hard to truly achieve this guarantee in a distributed system. There are some interesting mitigation approaches predating blockchain, but they do not completely solve the problem in a practical way.

Double-spending is solved easily by Bitcoin, as only one block is added to the chain at a time. Double-spending is impossible within a single block, therefore even if two blocks are created at the same time—only one will come to be on the eventual longest chain.



Bitcoin relies on the difficulty of accumulating CPU power.

While in a voting system an attacker need only add nodes to the network (which is easy, as free access to the network is a design target), in a CPU power based scheme an attacker faces a physical limitation: getting access to more and more powerful hardware.

This is also the reason malicious groups of nodes need to control over 50% of the computational power of the network to actually carry any successful attack. Less than that, and the rest of the network will create a longer blockchain faster.

## Ethereum

Ethereum can be thought of as a programmable blockchain-based software platform. It has its own cryptocurrency (Ether) which fuels the deployment of ***smart contracts*** on its blockchain.

Smart contracts are a piece of code stored as a single transaction in the Ethereum blockchain. To run the code, all you have to do is issue a transaction with a smart contract as its destination. This in turn makes the miner nodes execute the code and whatever changes it incurs. The code is executed inside the Ethereum Virtual Machine.

***Solidity***, Ethereum's native programming language, is what's used to write smart contracts. It is a turing-complete programming language which directly interfaces with the Ethereum blockchain, allowing you to query state like balances or other smart contract results. To prevent infinite loops, running the code requires some amount of Ether.

As the blockchain can be interpreted as a series of ***state changes***, a lot of Distributed Applications (DApps) have been built on top of Ethereum and similar platforms.

### Further usages of distributed ledgers

***Proof of Existence***—A service to anonymously and securely store proof that a certain digital document existed at some point of time. Useful for ensuring document integrity, ownership and timestamping.

***Decentralized Autonomous Organizations (DAO)***—organizations which use blockchain as a means of reaching consensus on the organization's improvement propositions. Examples are Dash's governance system, the SmartCash project

***Decentralized Authentication***—Store your identity on the blockchain, enabling you to use single sign-on (SSO) everywhere. Sovrin, Civic

*And many, many more. The distributed ledger technology really did open up endless possibilities. Some are most probably being invented as we speak!*

. . .

## Summary

In the short span of this article, we managed define what a distributed system is, why you'd use one and go over each category a little. Some important things to remember are:

- Distributed Systems are complex
- They are chosen by necessity of scale and price
- They are harder to work with
- CAP Theorem—Consistency/Availability trade-off
- They have 6 categories—data stores, computing, file systems, messaging systems, ledgers, applications

To be frank, we have barely touched the surface on distributed systems. I did not have the chance to thoroughly tackle and explain core problems like consensus, replication strategies, event ordering & time, failure tolerance, broadcasting a message across the network and others.

## Caution

Let me leave you with a parting forewarning:

You must stray away from distributed systems as much as you can. The complexity overhead they incur with themselves is not worth the effort if you can avoid the problem by either solving it in a different way or some other out-of-the-box solution.

> Don't get addicted to the buzz that comes with solving hard problems. If you're solving the wrong problems, your effort will be wasted. If you miss a chance to turn a hard problem into an easy one, your effort will be wasted. Find inspiration in progress, not problem solving.

. . .

***[1]***
Combating Double-Spending Using Cooperative P2P Systems, 25–27 June 2007—a proposed solution in which each ‘coin’ can expire and is assigned a witness (validator) to it being spent.

*Bitgold*, December 2005—A high-level overview of a protocol extremely similar to Bitcoin's. It is said this is the precursor to Bitcoin.

## Further Distributed Systems Reading:

*Designing Data-Intensive Applications, Martin Kleppmann*—A great book that goes over everything in distributed systems and more.

*Cloud Computing Specialization, University of Illinois, Coursera*—A long series of courses (6) going over distributed system concepts, applications

*Jepsen*—Blog explaining a lot of distributed technologies (ElasticSearch, Redis, MongoDB, etc)

. . .

Thanks for taking the time to read through this long(~5600 words) article!

If, by any chance, you found this informative or thought it provided you with value, please make sure to give it as many claps you believe it deserves and consider sharing with a friend who could use an introduction to this wonderful field of study.

*~Stanislav Kozlovski*

. . .

## Update

I currently work at Confluent. Confluent is a Big Data company founded by the creators of Apache Kafka themselves! I am immensely grateful for the opportunity they have given me—I currently work on Kafka itself, which is beyond awesome! We at Confluent help shape the whole open-source Kafka ecosystem, including a new managed Kafka-as-a-service cloud offering.

We are hiring for a lot of positions (especially SRE/Software Engineers) in Europe and the USA! If you are interested in working on Kafka itself, looking for new opportunities or just plain curious—make sure to

message me on Twitter and I will share all the great perks that come from working in a bay area company.