UNIVERSITATEA TEHNICĂ A MOLDOVEI

Distributed Systems

Alexandru Burlacu

Autumn 2020

FACULTATEA
CALCULATOARE, INFORMATICĂ
ȘI MICROELECTRONICĂ

FCIM

Intro & Course Description

Alexandru Burlacu

Autumn 2020

But first...

Congratulations!

You’re almost done. Just a little more suffering and you’ll be officially Engineers.

PR

Concurrency primitives + protocols

PTR

Concurrency w/ messages + streaming

PAD

Distributed systems and their perils

- **Topics** - first a quick recap of all we did up to this point, and then how to build distributed systems (services, DBs, infra), and the icing - What makes distributed systems really hard
- **Labs** - 1st A microservices-based system, 2nd add support for distributed transactions and other fancy stuff
- **Midterms** - two midterms, a lab (70%) + questions (3 Qs = 5 + 15 + 10)
- **Exam** - oral, 30 min preparation time, <16 min Q&A
- **Grading policy** - 10 is thresholded at 91 points, the rest are relative, following Gaussian dist.
- **Attendance** - Doesn’t matter. Just pass the exam and complete the labs on time

Something new

Lab projects will be done in groups of 3-4 people, everyone will be responsible for their part of the project. Lab points will be assigned by team members from a points pool.

A. Concurrency and Messaging patterns recap
B. Components and patterns of a distributed system
C. Some high level protocols/means of communication
D. A short intro into databases
E. Some useful theory
F. Distributed transactions
G. Maybe some more nice things

[Recap] Concurrency

Alexandru Burlacu

Autumn 2020

Many Flavors of Concurrency

Remember it from last semester?

Alexandru Burlacu

Autumn 2020

Actor Model, CSP, and friends

Recall from last semester our discussions about different flavours of message-passing concurrency. Actor model? Communicating Sequential Processes? Reactive streams? Anyone?

And then more high level stuff like Message Brokers and all the related patterns? Well, we'll need all of it during this semester.

Remember: for distributed systems asynchronous message passing is the way to go! Only when it isn't.

Total recall: Actor Model

Actors:

- Have identities and addresses
- Can communicate if they know the receiver
- Use fire-and-forget type of messaging
- Are sequential in nature but communication enables concurrency
- Are good to represent isolated/safe state, and hiding the information where this state resides
- Are cumbersome when it comes to composition, and generally are quite opinionated
- Are favored among Erlang/Elixir/Scala people. Also available for others. i.e. Project Orleans from MS

Total recall: CSP

Communicating Sequential Processes are:

- … anonymous and need a channel to communicate
- … formally defined and allow proving some system properties
- … communicating synchronously (with sized channel - async, but not well formalised)
- Are good to represent tasks/actions
- Favored among Golang/Clojure and others too.

Total recall: Reactive Streams

Reactive Streams are:

- … good when it comes to representing streams of events/data
- … especially useful when backpressure is inherent in the system
- … available either us separated projects, like in Elixir, or as RxSomething library
- … liked by front-end/mobile devs
- … becoming popular among microservices devs

[Recap] Messaging patterns

Alexandru Burlacu

Autumn 2020

Messaging Patterns

Using a message broker and some other messaging middleware it is possible to build highly scalable, decoupled systems. What's not to like.

In fact, this is the first step towards distributed systems.

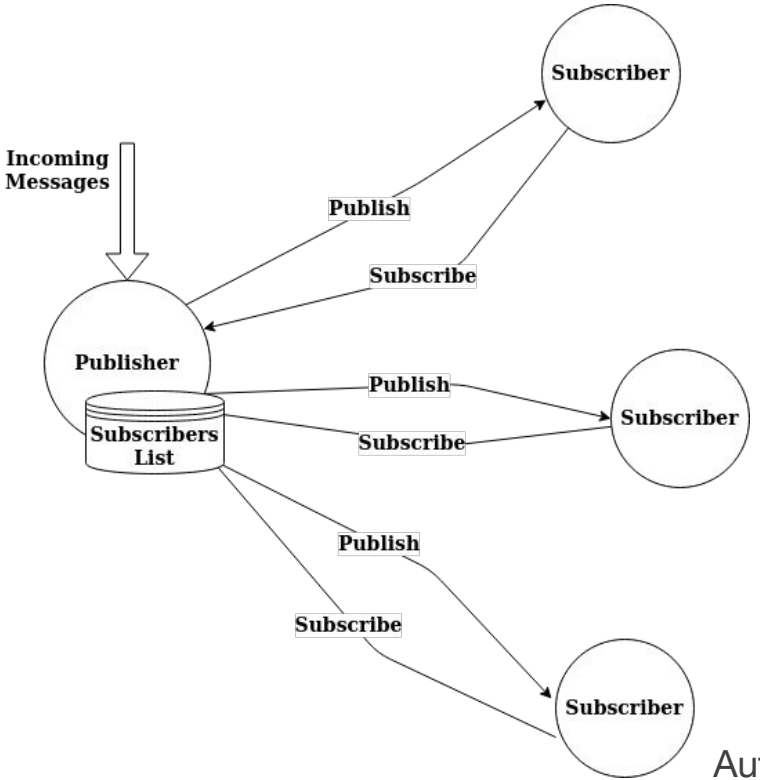
Let's recall first one of the fundamental patterns of messaging communication, **PubSub**.

Publisher-Subscriber

A quick reminder of what PubSub is:

You have a set of message/event creators, let's call them **publishers**, and another set of entities that are interested in some, or all of the messages/events, these are **subscribers**.

Subscribers can subscribe to publishers anytime they want and receive updates as soon as possible.



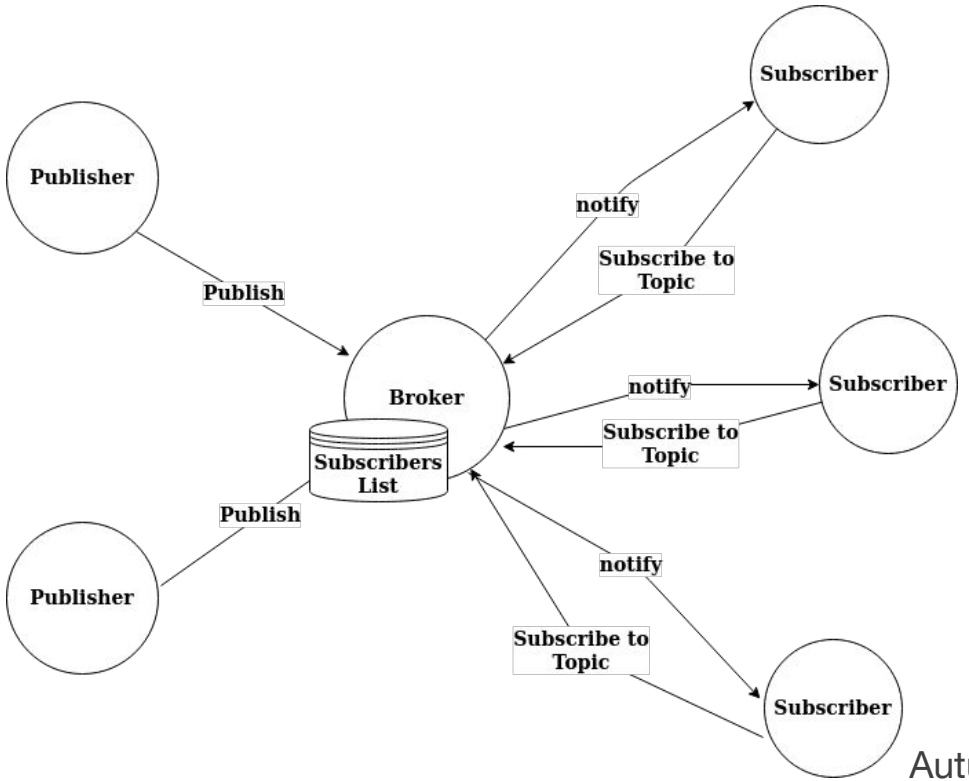
Messaging Patterns

PubSub is cool, no doubt, but sometimes, when there are many entities involved, it can be remodeled to decouple things better. Enter the **Message Broker**.

And while recalling, also think about how a message needs to be (major hint: **self-contained**) and what tricks there are to make the message broker resilient to any kinds of mischief (another major hint: **durability and persistence**).

Message Broker!

Enter the Message Broker!

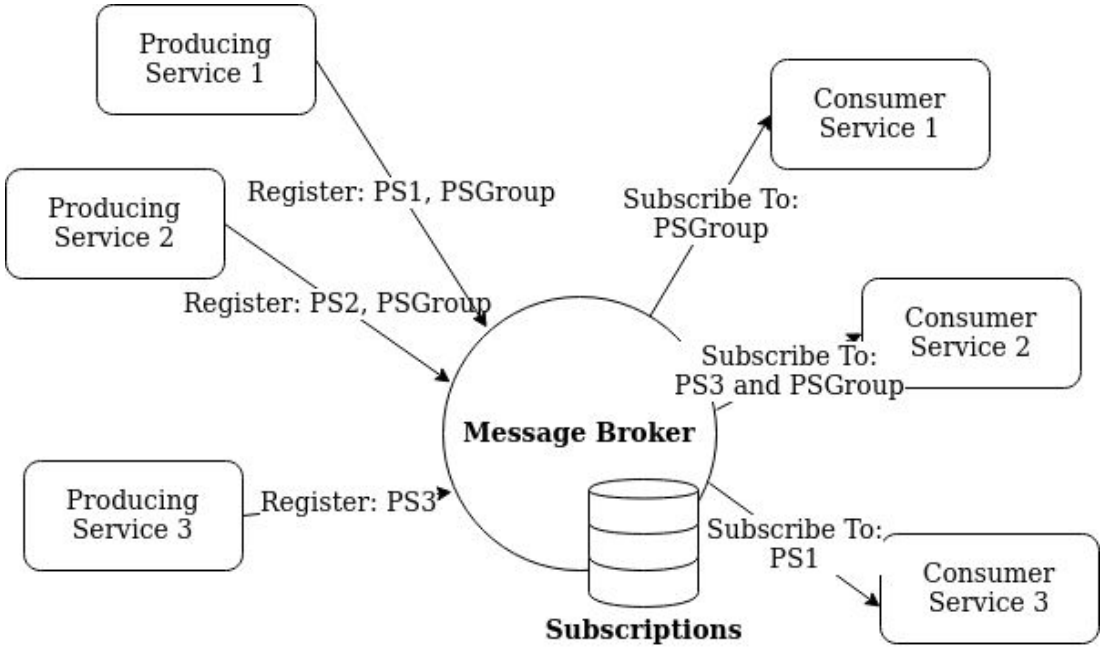


Message Broker!!

Topics for Message Brokers

You can create new topics either on the producer or consumer side.

What if the topic is generated on producer side and there are no consumer services to read from it?



Can’t put better than this.

Act 1: The (not so) hard things

Alexandru Burlacu

Autumn 2020

How do we build Netflix?

Or Google/Amazon/Facebook/Insta/whatever

Alexandru Burlacu

Autumn 2020

So how do you do it?

Well, have you heard about **System Design**? Or maybe the feared **System Design Interview**?

This is actually a very common type of question.

Basically, you first ask clarifying questions, then state your assumptions, and based on it make your best possible decisions (read tradeoffs). So, kinda like our last 2 exams.

And it also involves a lot of drawing of rectangles, arrows and circles.

Ok, seriously, how??

You need to know
(1) your components,
(2) their trade-offs,
(3) common distributed systems patterns,
    (3.1) their trade-offs too, and finally
(4) usually knowledge about databases and generally data-centric systems is needed, like analytics engines, storages, databases.

We will talk about all this, except from going into details for (4).

P.S. I wasn't joking in the previous slide, you do need to know all that

Reactive Manifesto - again

Reactive Manifesto - again

Recall Reactive Manifesto, namely its four components.

Recall, it proposes a new mindset of system design, based on asynchronous **message passing** for communication, allow decoupled entities to scale up and down, or be **elastic**, also, because of the 2 properties, such systems would be **resilient** against failures, allowing components to fail and recover independently, and as a result be **responsive**.

You should really$^{k,\ \text{where}\ k > 1}$ read it: https://www.reactivemanifesto.org/

Main concerns: Transparency

Transparency, what??

Transparency in the context of human-computer interaction means possibility to alter the internal behaviour and/or implementation without changing the external interface. In other words, **transparency is** actually **invisibility**.

An example of transparency - when you keep data in your Google Drive you never notice that it might have changed the server where it is placed. This is both **location and migration transparency**.

Main concerns: Transparency

Some most important types of transparency, for dist. systems at least:

- Access transparency (uniform access)
- Concurrent transparency (true concurrent access)
- Failure transparency (if something breaks, everything else works)
- Location transparency (it can be anywhere)
- Relocation transparency (it can be moved anywhere while working)
- Replication transparency (it can have copies)
- Fragmentation transparency (it is made of fragments)
- Persistence transparency (it can be either on disk or in memory)

Yet the user doesn't see it/care about it.

Also, `it` refers to both data and processes.

Main concerns: Transparency

Into some more details, if you need:

- **Access transparency** – Regardless of how resource access and representation has to be performed on each individual computing entity, the users of a distributed system should always access resources in a single, uniform way.
- **Location transparency** – Users of a distributed system should not have to be aware of where a resource is physically located.
- **Relocation transparency** – Should a resource move while in use, this should not be noticeable to the end user.
- **Replication transparency** – If a resource is replicated among several locations, it should appear to the user as a single resource.

Main concerns: Transparency

Into some more details, if you need:

- **Concurrent transparency** – While multiple users may compete for and share a single resource, this should not be apparent to any of them.
- **Failure transparency** – Always try to hide any failure and recovery of computing entities and resources.
- **Persistence transparency** – Whether a resource lies in volatile or permanent memory should make no difference to the user.
- **Fragmentation transparency** – Regardless of what a resource is made of (fragments), the users of a distributed system should always access resources in a single, uniform way.

Transparency - Quiz Time!! (for practice only)

Q1. If you try to access a file on Google Drive via a link, it's not working, and notice that the valid link has changed, what kind of transparency the system is lacking?

a) Location b) Relocation c) Access

Q2. Say, in order to register on a social media service, your request is processed by 3 microservices before giving you a response, and of course you don't notice that. What kind of transparency is this?

a) Relocation b) Failure c) Concurrency

Q3. If I have a distributed database that keeps shards of tables on different servers, and uses optimistic locking, which of the following types of transparency is not valid, given the description:

a) Failure b) Fragmentation c) Concurrency

Main concerns: Scalability

One of the primary reasons we do distributed systems is to increase some performance metric(s) of the overall system, i.e. to scale it.

Scaling:

- Vertical (scaling up), that is make hardware faster, optimize code
- Horizontal (scaling out), that is add more machines to the party

Vertical scaling can get you only so far, so distributed systems are about Horizontal Scalability.

Main concerns: Scalability

Whenever we have horizontal scaling we need to distribute work among workers.

Load balancing is used solution. For example, you have 3 services, how do you distribute tasks between them, say connections?

- Round robin (easy, dumb, sometimes bad)
- Least connected (quite easy, better, sometimes not enough)

What if you have sessions and want each user to be “assigned” to just one server, and not float between all of them?

- Sticky sessions (use hashing, sometimes not good, other times just the right thing)

Main concerns: Scalability

Modern load balancers usually work on L4 of OSI, that is TCP or UDP, so they are oblivious of application specific requirements.

That's where L7 load balancers come into play.

An example of L7 load balancer is one that can use cookies to provide sticky sessions, or knows that it just received a HTTP/2 connection and not HTTP/1.1.

An example of L4 load balancer would be a much simpler one, like the one used in Kubernetes, or NGINX.

Why L4 load balancers? It's simpler and therefore faster.

*More load balancing algs. here: https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques/

Main concerns: Scalability

Another strategy for scaling a system is to use caches. By now you know what a cache is, so here are some ways caches can increase the performance of your application:

- **Client-side/Browser cache**, as to not even touch the network for some data
- **CDNs (content delivery networks)**, as to not touch the server for big binary files, like minified JS, images, or other media content
- **Server-side cache**, like Redis, to cache some server results, when frequently queried

[Top to bottom] Hardest to invalidate, cheapest, fastest to deliver

Main concerns: Scalability

Client-side/Browser cache is one of the cheapest ways to scale a service. In order to do so effectively, you need to know 2 HTTP Headers, namely **`Cache-Control`** and **`ETag`**. But there are more, like **`Vary`**.

Let's start with **`Cache-Control`**. Depending on how it's set it can either not cache at all (**`no-store`**), or cache (**`private`**) but only on the browser and so on. Also, depending on how it is set, it can specify maximum age for a resource, or whenever the client will accept a stale version of the resource, and optionally how stale can it be.

```
Cache-Control: private, max-age=604800
Cache-Control: public, max-stale=3600
```

More info here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

Main concerns: Scalability

Now, in regards to **Client-side/Browser cache** we also mentioned **`ETag`**.
Entity Tag, or ETag for short, is a way of checking whenever the resource has changed and therefore further helps in caching. It works like an optimistic concurrency control system.

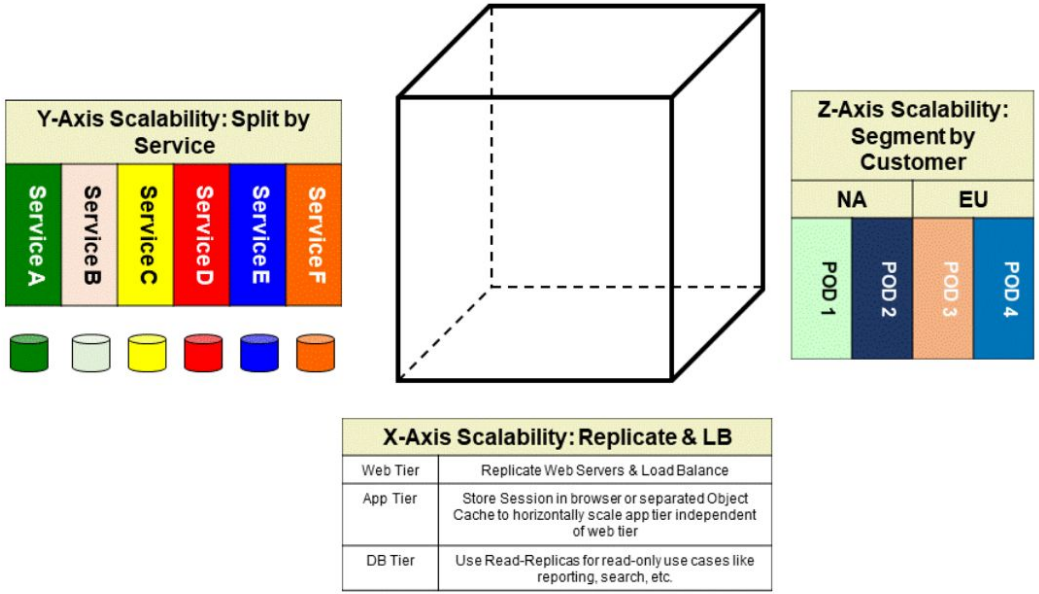
Say, you have stored in browser some JSON from a server, this JSON doesn't change frequently, and it happens approx. every hour or so, but sometimes it can change even after 3 hours. The devs used a cache control header to set it's maximum age to 3600 seconds, and allowed staled content.

With the help of an **`ETag: <the etag>`** that was received, the browser will send a new request with the **`If-None-Match: <that same etag>`** header, and in case the ETags match, the server will return a very small response with **304 Not Modified** status.

More info here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

Main concerns: Scalability

In a more formal, and complete manner, we can describe distribution, scalability, availability, and even fault tolerance (later on these) through the prism of **“The Scale Cube”** or “AKF Cube”



*Source: https://akfpartners.com/growth-blog/scale-cube

Main concerns: Scalability

The Scale Cube can be used to guide scaling initiatives within an organization. Keep in mind that scaling on Z- and especially Y-axis is more often than not also an organizational challenge (read, hard to implement).

In case you didn't quite get the diagram, find the explanation of each axis below.

X-axis: clone/replicate data/processes
Y-axis: functional decomposition, as in microservices
Z-axis: sharding, or data/process splitting by some attribute, could be location, user type or something else

More info here: https://akfpartners.com/growth-blog/scaling-your-systems-in-the-cloud-akf-scale-cube-explained

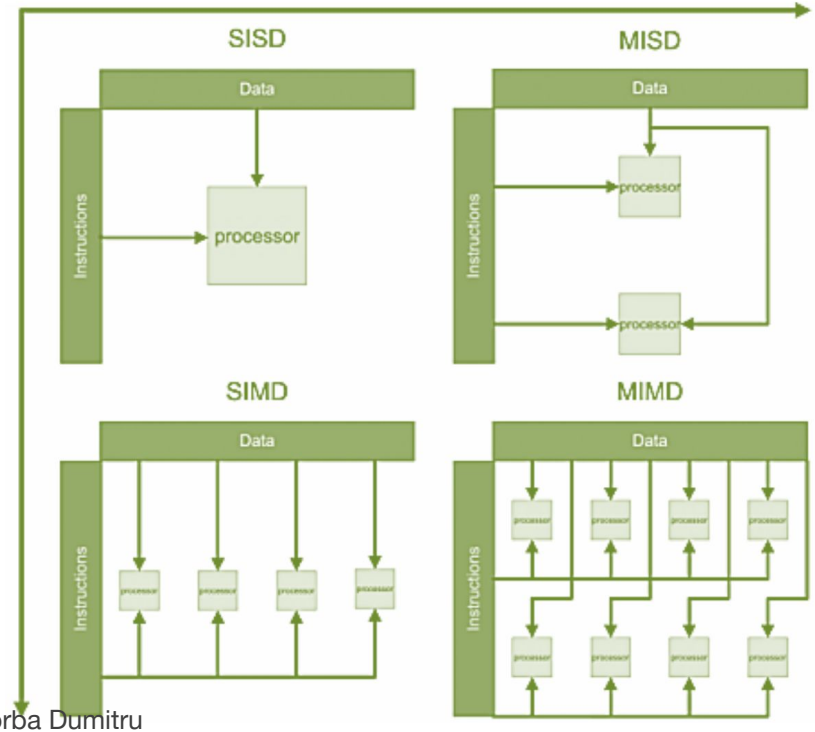
Main concerns: Scalability

This is Flynn's Taxonomy (as a reminder).

SISD - single instruction, single data
SIMD - single instruction, multiple data
MISD - multiple instruction, single data
MIMD - multiple instruction, multiple data

Also consider:
SPMD and MPMD, where instead of Instructions (I), Programs (P) are considered, relaxing the **lockstep** requirement.

Note: we relax the lockstep requirement



Source: PAD course, "Distributia: spatii de decentralizare", conf. univ. Ciorba Dumitru

Main concerns: Scalability

Recall the **Universal Scalability Law**. It states, and reasonably so, that as the number cores/machines computing something in a parallel/concurrent fashion grows, at some point not only the system will scale slower, but in fact the performance will degrade.

What does it mean for us?

To achieve maximum scalability, we need to design systems that (1) seldom need to communicate and (2) can be reasonably distributed among as many workers as possible.

Scalability - Quiz Time!! (for practice only)

Q1. Say, we have a site that even if we don't have access to the Internet some part of it can still be loaded in our browsers, given it was recently accessed (<1day). What kind of caches were used?

a) Client-side b) CDN c) Server-side

Q2. What are the benefits of using a CDN?

a) Lower latency b) Offloading main servers c) Cheapness

Q3. Why using a cache increases scalability of a system?

a) Minimizes computations b) Is closer to the user c) Speeds-up computations

8 misconceptions about Distributed Systems

Distributed systems are treacherous. Because a lot of issues are already dealt with (kinda) and abstracted away, programmers tend to fall for the following fallacies:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous.

Consider reading this: www.rgoarchitects.com/Files/fallacies.pdf

8 misconceptions about Distributed Systems

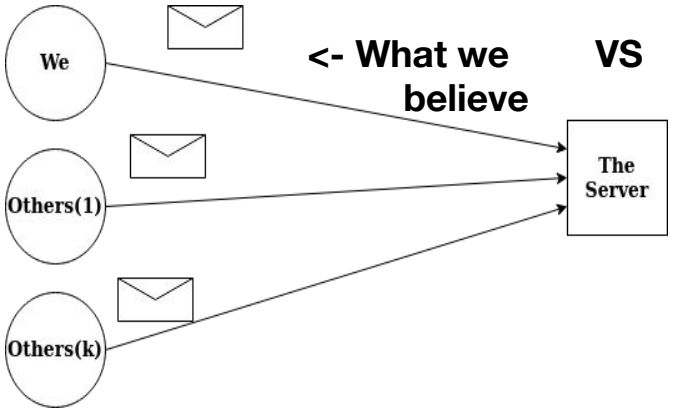
As a result we can have the following issues:

1. **The network is reliable:** Never, things break, and not taking it into account will cause a lot of *incredible* network bugs. See xkcd.com/2259/
2. **Latency is zero:** Ofc, and unicorns are real. We are limited by the speed of light, not taking it into account will lead to premature timeouts, and bandwidth inefficiencies in general
3. **Bandwidth is infinite:** Nah, and if you think so good luck explaining why the network is congested
4. **The network is secure:** Nope, and assuming so exposes the system to great risks
5. **Topology doesn't change:** It does, and as a result it affects latency and bandwidth
6. **There is one administrator:** Configuration and policy conflicts, hidden issues
7. **Transport cost is zero:** Projects may overrun their budgets
8. **The network is homogeneous:** See first three issues

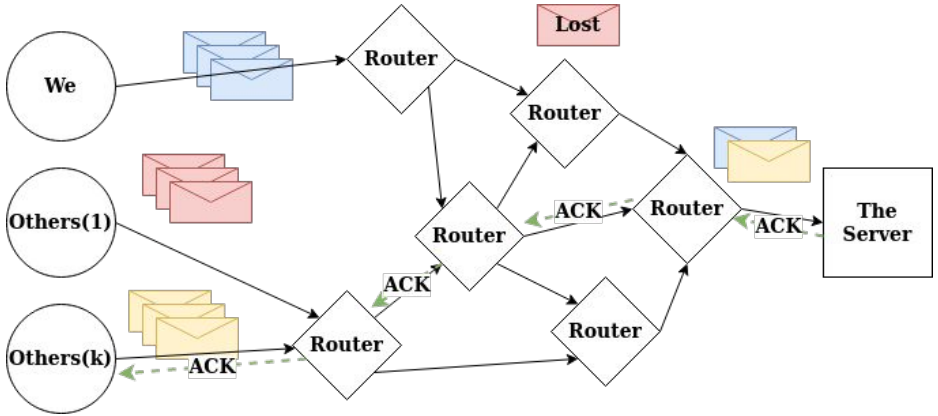
We treat the network wrong

Usually we think about computer networks being like direct connections between parties. Even if we know that there are routers and switches and cables with limited capacity between us and the destination, we still somehow believe we connect directly over a dedicated channel.

!!!And that's a wrong mental model!!!



<- What we believe VS Reality ->



Main concerns: Resiliency

Now, assuming we dealt with scalability, how to we ensure that when things get stormy, we are still up and running?

For that we need to think about **resiliency** of our systems.

Resiliency means that it can keep running despite issues and failures. Therefore, a resilient system is **not failure-proof** but rather **failure-tolerant**.

Note that ideally we would like systems that are failure-thriving, or antifragile, but in order to achieve this we need a good development and operating workflow, and that's beyond our scope.

Main concerns: Resiliency

Just as with scalability, one of the primary tools we can use to ensure our systems are failure tolerant is having redundancy in our systems, i.e. **replication**.

How does this work? Easy - we just have copies of our data and services, such that in case we lose particular instances, we still have our “backups”.

For example, we can replicate our databases in case some database instances fail.
Or, we could have multiple copies of our services, to ensure that if some server dies, the whole system continues to work.

This strategy is also known as **High Availability** setup.

Main concerns: Resiliency

But now the question is, **if we fail, how does our backups proceed?**

For that we need failover strategies, and normally there are two:

- **Active-active**: have both the primary and secondary system running, if primary fails, immediately direct the traffic to the secondary
- **Active-passive**: have only the primary system running, if it fails, launch the secondary system and start receiving all the traffic on it

… and of course there are trade-offs. Can you tell what are they?

Main concerns: Resiliency

Well, active-passive failover is cheaper, but slower to react and with more points of failure. Active-active is more expensive, but much faster and actually a safer bet.

But replication and failover are not the only ways we can achieve HA/Resiliency.

We also need to ensure we know when things go off-rails, as they happen, even if for the user its transparent.

For this we need **observability** in our systems.

Main concerns: Resiliency via Observability

Observability, what is it? Basically, the property of a system to be understood, or how well can one infer its internal state from external outputs. It's a spectrum, and depending where on it our system stands, we can use monitoring and alerting more or less efficient.

In other words, if a system is observable we can understand what is happening within it from its outputs. We need to design observable systems. And to aid us in this we have **logs**.

Logs are the bread and butter of monitoring and observability.

With good logs we can do performance monitoring, incident analysis, or debugging, and tracing.

Main concerns: Resiliency via Observability

How our logs should be?

- **Hierarchical**: we need to respect the distinction between DEBUG/INFO/WARNING/ERROR levels, and not to crowd the system with WARNING logs when INFO or DEBUG are more appropriate. Not-to-crowd also refers to how much information a log contains.
- **Filtrable**: logs are meant to be analyzed. Make them as searchable as possible. Consider formatting them as JSONs, and don't abuse nesting.

For performance monitoring consider DEBUG logs that contain execution time of your code. If you add correlation IDs to these logs and pass them between services, now you have a form of **tracing**. Tracing will help you find bottleneck services, and sometimes even aid you in debugging distributed systems.

Main concerns: Resiliency and Reliability

Alright, we figured out how to survive failures, but it's always a good idea to not test our fortune, and design systems with a lesser chance of failure. **Reliability** of our systems is still a concern.

There're many ways in how we can make our systems more reliable, but here are 3 patterns for it:

- Retry policies
- Circuit breakers
- Bulkheads

Main concerns: Resiliency and Reliability

Imagine, service A tries calling service B for some data, and after a short timeout, it fails. Most likely service B had some transient issues or the network is slow. What should service A do?

A lot of the times, the reasonable answer is to **retry**. Some popular retry policies are: retry with a delay and exponential backoff.

The simple retry is, well, simple. But sometimes it is more reasonable to use the exponential backoff, in order to minimize the number of retries, therefore allowing the service to do other stuff instead.

Basic example of an exponential backoff: service A after initial fail to access B, waits for 2 seconds and calls it again, if it fails again, A will wait for 4 seconds, and if failure occurs again, for 8 seconds, and so on...

Main concerns: Resiliency and Reliability

… and on, and on. If you're thinking that it is stupid to retry on and on again ad infinitum like this is stupid, you're right.

That's where **Circuit Breaker**s come into play. You can think of them as an extension to the retry policies.

Circuit breakers are meant to limit the number of retries in cases where the failure is not gone quickly. You can think of it as some sort of fail-fast. In case of services A and B, circuit breaking could be triggered in case B is not responsive on numerous occasions, possibly due to a complete failure of the service.

Waiting for this kind of issues to be fixed is unreasonable, thus defaulting to an error is prefered.

Main concerns: Resiliency and Reliability

And finally, **Bulkheads**.

A bulkhead actually means an upright partition separating compartments, like in boats and ships. And modern ships use bulkheads not just for increased structural strength, but also to be able to withstand partial floodings, separating flooded compartments from the dry ones.

In system design a bulkhead is more or less the same. It limits the amount of resources given per service/functionality/task and isolates it, thus in case some issue happens either on client or server side, we won't have all the resources consumed on it, only a predefined amount.

First things (almost) first: Services

We are going to need to define our services, be they micro- or macro-; then, we will need to define their APIs.

Most of the time it is recommended for a single microservice to have it's own DB. Sometimes, sharing is ok. For example when having very related services. Or for legacy reasons. Following is a list of service-database relationship, from best case to worst:

- Database-per-Service
- Schema-per-Service
- Tabel-per-Service

Consider reading this: https://cloudncode.blog/2016/07/22/msa-getting-started/

First things (almost) first: Services

Having microservices it is common/recommended to use a so-called **Reverse Proxy** in front of a microservice. Why?

Consider reading this: https://cloudncode.blog/2016/07/22/msa-getting-started/

First things (almost) first: Services

A reverse proxy is a dedicated server that provides multiple benefits at the cost of increased complexity.

NGINX is a reverse proxy, what does that mean?

It means that NGINX can be used for load balancing, static resource serving (HTML/CSS/Images), traffic compression, but also failover and a lot of other stuff.

So it’s a load balancer on steroids.

First things (almost) first: Services. How to adopt?

Before we dive any deeper, how do you adopt microservices?

First - most of the time you should never start with microservices! Because this way you're violating the YAGNI principle. Maybe the project will fail and all the complexity wasn't necessary? Or maybe, the project will fail **because** it was complex and too slow to validate. Start with a well designed monolith.

Software should the thought of as growing, rather than being designed. Change is the only thing that's certain.

First things (almost) first: Services. How to adopt?

Before we dive any deeper, how do you adopt microservices?

Now let's say that you built your monolith, and it was successful, and you have to scale using microservices. How?

Enter the **Strangler pattern**.

The idea is rather simple - put a façade in front of your system, and gradually split the monolith into multiple services, the façade will ensure that the users won't notice.

More info: https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler

First things (almost) first: Information hiding

Once we have APIs and our services up and running, the last thing we want is for the client to be concerned about the existence of many services within our system.

Solution: use an **API Gateway**
or its more advanced variant: **BFF (backend-for-front-end)**

First things (almost) first: Information hiding

An API Gateway is a centralised point of access for all/most of your services.

Think of it as the Façade pattern, only for the microservices world. Basically it's a single point of access, sometimes it might contain additional logic like authentication and maybe some cross-service logic, like simple aggregations.

Normally the later is recommended to be done on separate microservices, but with a small system this could be acceptable.

First things (almost) first: Information hiding

Now, BFF or Backend-For-Frontend is basically an API Gateway, tuned for a specific type of client. Initially used at Spotify, it is common to have BFFs for mobile clients, web clients and sometimes dedicated BFF for the API.

BFFs allow to even have multiple types of APIs/protocols for different clients.

For example for the web API HTTP/2 can be used, maybe in combination with GraphQL. While for an Android client, gRPC or Thrift RPC technologies can be used.

The downside of a BFF - well, there’s more code to maintain.

The unnamed hero - Infrastructure: Service discovery

There's an issue with splitting a monolithic codebase into microservices. How do we locate the stuff we need?

In local systems we know for sure what is where. Not so much in case we run things on multiple servers. Now, take into account that servers might fail and be restarted, we could scale some instances and not others, when traffic surges, and we should also be able to migrate things.

With this dynamic context, we can't rely on hardcoded addresses for our services.

What do we do?

The unnamed hero - Infrastructure: Service discovery

We do what is called **service discovery**. It's not a new thing. Remember DHCP? Also partially the problem of service discovery is solved by DNS and ARP protocols. But all these are related to low-level, networking technologies. We are now in the realm of services.

So the basic idea is the following, we have a dedicated registry where we keep all the information about how to access a given service. When a service launches, it registers (either itself, or via some 3rd party) to the service discovery registry.

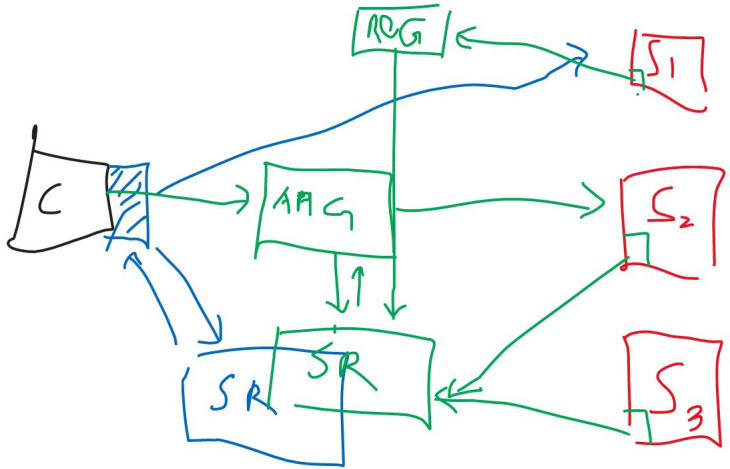
Now, how do we find the concrete address of the service?

There are 2 ways: client-side service discovery and server-side service discovery.

The unnamed hero - Infrastructure: Service discovery

Client-side SD (blue) assumes we have a service registry (SR) aware client that first asks it about the location of the service of interest and then makes the request.

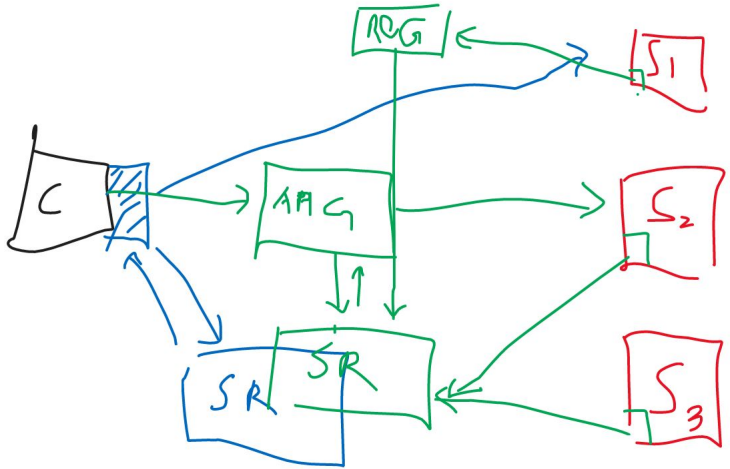
Server-side SD (green) uses an intermediary, for example a reverse proxy, to perform the duties of the SR-aware client. Client has to just call the service by name, the intermediary will find the address of it and forward the call.



The unnamed hero - Infrastructure: Healthchecks

Whenever we do client-side or server-side SD, how do we make sure that the address we're getting is of a server that's up and running?

For that we have **healthchecks** (caller checks if server is up), or **heartbeats** (server notifies the caller that it's up), depending how the information is sent. In our case, the caller is SR.



The unnamed hero - Infrastructure

Service meshes are on the forefront (read hype) of microservice infrastructure. Remember reverse proxies?

Imagine every service gets a reverse proxy through which every request goes through, and it is now possible to monitor all the traffic, use circuit breakers and request pipelining whenever possible/necessary.

Not only that, but depending on the exact tool one uses, it is also possible to route traffic, do autoscaling of the instances, for load balancers/reverse proxies are already in place.

Even chaos engineering is now possible, by injecting failures into the system.

An example of service mesh is **Istio**, there's also **Consul** which can be adapted for this.

Communication means: REST

REST or REpresentational State Transfer is one of the most widely acknowledged (not as much properly implemented) interaction patterns used when it comes to high level communication.

In fact, REST is an architecture for networked systems, that is defined by its constraints.

There are 6 main constraints for REST:

- Client-server architecture
- Statelessness
- Uniform interface
- Cacheability
- Layered system
- Code-on-demand (optional)

Communication means: REST

Before we dive into REST constraints, first we need to understand why it exists and even what does it mean.

Let's dissect the name - representational state transfer.

- **Representational**: a RESTful service will send a negotiated representation of the resource the client wants. For example, we won't show the Java code or some binary representation of the User, rather, a JSON with content about it.
- **State**: REST APIs must send the state of the session within the request, and never store it on the server.
- **Transfer**: REST APIs transfer entire data back to the client, not some reference to it. You get a JSON not some ID by which to modify the object somewhere on the server.

Communication means: REST

So what does all of these constraints mean?

Client-server architecture means there's a client, requesting something and a server that provides specific services. Making this distinction goes naturally with how the Web is built.

Given that REST was designed to handle the requirements of a decentralized, always evolving Internet, performance is crucial. Also, given the client-server interaction, where clients are considerably more numerous compared to servers, **statelessness** provides easier scalability, thus increased performance.

Also, stateless servers aid in debuggability, by making it possible to understand the state of the system by just looking at the request.

Communication means: REST

Uniform interface means that whenever we are trying to access a document, binary file, some dynamic content or even write to some location, we are provided with an easy, uniform interface.

REST relies on URIs for this, and the author, Roy Fielding, recommends using *resource* names for the URIs, not actions done on them. A resource in REST parlance is the entity we want to interact with. Again, it could be something materialized, like a file, or dynamic, like the time of day at the moment of call.

Coming back to performance and scalability concerns, and given that we already have stateless servers, **cacheability** is another desired property for a large scale information distribution system, which REST is primarily designed for.

Communication means: REST

Finally, we have the the possibility to make our systems **layered**, for example given a request, it might go through some firewall, then gateway, than one service, and then another. All of this is done in a easy, transparent way.

In a way, layering, in combination with the other constraints so far, makes it possible to interpret REST as another architectural pattern, **uniform pipes-and-filters**.

Layering is necessary to keep the complexity of the systems at bay.

Communication means: REST

And finally, the last, optional constraint, **Code-on-demand**.

It is optional because of the fact that its non-trivial to implement, and not always necessary. Casually, code-on-demand is implemented as **HATEOAS** (Hypermedia as the engine of application state), but isn't necessarily HATEOAS.

Code-on-demand is in fact a mobile code paradigm that states that it is possible to send the client additional code to extend its functionalities or further reduce the required knowledge to interact with the system. Sounds familiar?

Yup, code-on-demand is basically JS, or for older web applications - Java Applets.

Communication means: Not just REST

By now maybe you know that REST, and generally HTTP, is not the only way to connect distributed services. An entirely different paradigm are **RPC**s and Remote Objects.

RPC, or remote procedure call, sometimes, more in regard to Remote Objects, it is known as RMI, remote method invocation, is a way to call operations that are location transparent. RPCs were once proposed as a more abstract way to build software, but its scope was its own peril.

Abstracting away the fact that a call might be done using the network, thus taking orders of magnitude more time to complete is a bad idea, that's the reason it felt out of grace, until now.

Communication means: Not just REST

More recently, RPC re-entered the scene as a more performant, even if less flexible alternative to REST. Prominently, new kinds of RPC like gRPC and Apache Thrift are used by big corporations like Google and Facebook, respectively, when performance is important and developers are in control of both clients and servers.

The reason why new generation of RPCs are more successful than the old ones, like CORBA and SOAP protocol, are because they are more lightweight, API-wise, and they do distinguish between local and remote calls.

Communication means: Not just REST

Finally, let's discuss **Remote Objects**, as the next step from RPC. Disclaimer, the approach is not so popular anymore, with a but.

So, Remote Objects' basic idea is: to abstract away the location, and possibly the number, of an object, in OOP sense. If RPCs abstract away the location of a function/operation, Remote Objects abstract away the location of an object (operations+data).

Cap'n'Proto is a modern revision of this approach, while CORBA and DCOM are some old variants.

Communication means: Not just REST

Previously, I said that remote objects are not used anymore, at least not as much, with a but…

The “but” is, they actually are, but in the form of SDKs for different services. Basically, if you provide an OOP SDK for some remote service, you’re halfway there. Of course, this is an oversimplification, but it helps understand that it’s not such a dead idea.

Remote objects are usually replicated and use peer-to-peer communication to achieve consistency.

Communication means: SOAP and REST

SOAP is known for… killing bacteria off your hands.

Jokes aside, SOAP, or Simple Object Access Protocol, is a standard for RPC communication via HTTP (mainly) and using XML schema definition and for transport too.

Basic idea is that using a SOAP framework, and having the schema for your API in WSDL (Web Service Description Language), it can generate your stubs + input validation, and all you're left with is to implement controllers.

SOAPs benefits are compatibility with current Internet stack, and transport and language independence made it a popular choice, even if outdated today. The usage of WSDL was actually considered not so good, because it became very verbose.

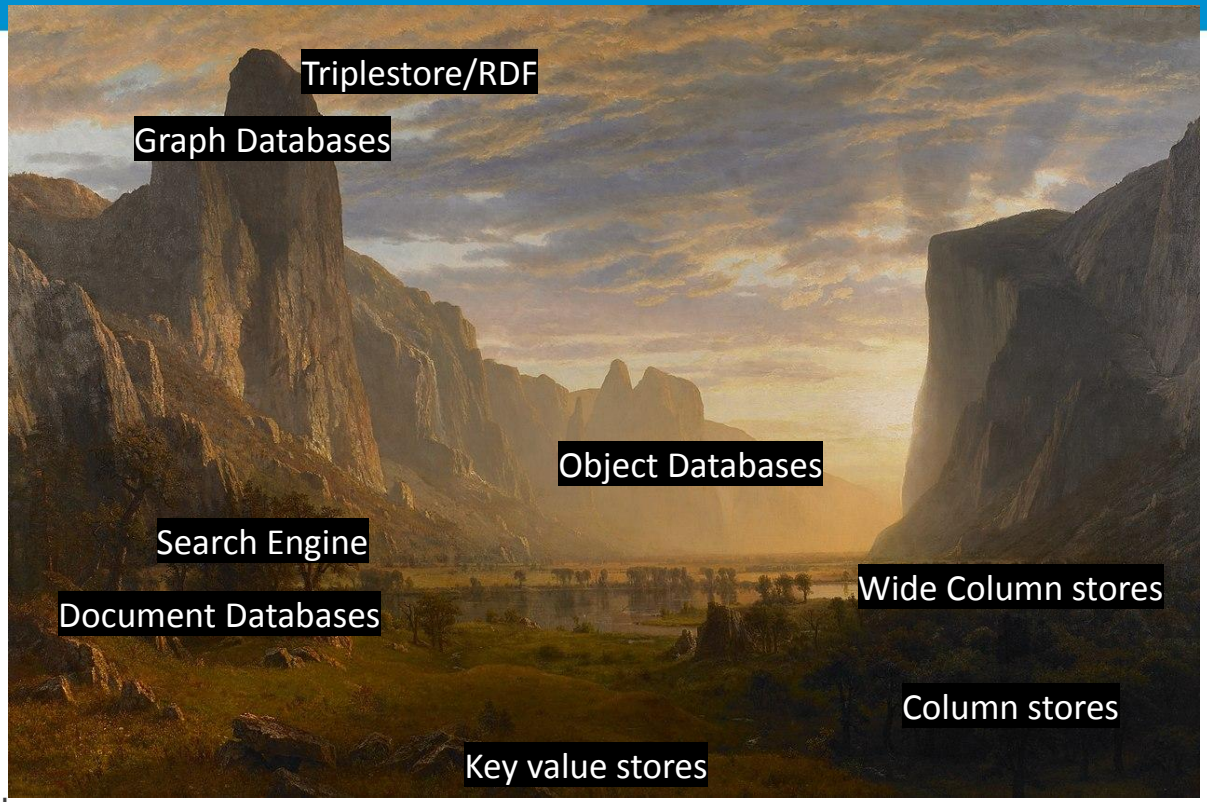
Communication means: SOAP and REST

Even if SOAP is mostly obsolete today, and people prefer REST, some ideas were refurbished. Like WSDL. As Swagger/OpenAPI.

Don’t get me wrong, Swagger was primarily developed for API documentation and testing, but, it is possible to use it to generate client and server stubs + validation + models + security (partially).

Swagger became popular because of its optionality. You don’t have to use it, neither you must use all the possible features it can give you. Besides, it’s not using XML, and for a lot of people this is a plus too.

The takeaway: even if the technology failed, doesn’t mean that some of its concepts were bad.



*Painting: *Looking Down Yosemite Valley, California*, Albert Bierstadt

Crash course into (NoSQL) DBs

NoSQL is like the painting above, a lush, beautiful and diverse landscape. NoSQL basically means everything that is not SQL/Relational.

Again, there are multiple NoSQL types but the most basic one is a **Key Value Store**, like Redis, or your good ol' HashMap. KV Stores are quite diverse in practice, ranging from simple ones like Memcached used for caching, and up to Riak, Apache Ignite and Aerospike which provide some consistency guarantees (Riak) and even support ACID transactions (Aerospike) and joins (Apache Ignite).

KV stores are also very performant but lack in functionality, compared to other types.

Crash course into (NoSQL) DBs

Then, there are **document databases**, like Mongo and Couchbase, and some would say Elastic also belongs in here, but more on that later.

Document Databases can be thought as more advanced KV stores, with the difference that for a KV store the content of the database is not important/relevant/used, whilst for document ones, the database is aware of its contents and can provide richer functionality based on this.

Basically with a canonic document database it is possible to query and filter based on the content of the document and its structure, where for a KV store this possibility is limited.

Crash course into (NoSQL) DBs

Elastic was mentioned as somewhat a document database. In principle it could be, because it can store information schemaless, is aware of the structure of documents and can query and filter based on that. But Elastic is actually a **search engine**.

Tools like Elastic, Sphinx or Lustre, or Solr, are using a very special data structure, called **inverted index**, to be able to do full text searches. How does this work?

Crash course into (NoSQL) DBs

A simple index will look something like this:

```
{
"file1.txt": "hello darkness, my old friend",
"file2.txt": "I've come to talk with you again",
"file3.txt": "because a vision softly creeping",
"file4.txt": "left its seeds while I was sleeping",
"file99.txt": "I was sleeping all day with a deadline running up"
}
```

While an inverted index for the same dataset will look like this:

```
{
"hello": ["file1.txt"],
"I": ["file2.txt", "file4.txt", "file99.txt"],
…
"sleeping": ["file4.txt", "file99.txt"]
}
```

Crash course into (NoSQL) DBs

Before we touch the big guns, graph and object databases, there's one more we need to discuss, **column stores**. Such technologies as Druid, HBase and Cassandra are the open source form of column databases. So what's so special about them?

Let's dissect what are they made of.

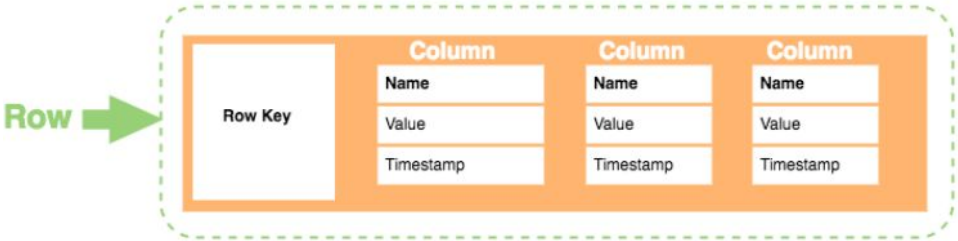
First, they are, just like document databases, an improvement over key value stores. But the difference is how the data is stored in there. Every "key" in such a database is the column name, while the value is a collection of values for the given column. As a result, these databases are blazing fast when it comes to reading speeds. They are also very fast for analytic queries.

Among other benefits, they are very easy to apply compression to.

Crash course into (NoSQL) DBs

Remember I told you Cassandra and HBase are column stores? I lied, a bit, technically.

They are **Wide Column Stores**. Which means they combine both row-based and column based approaches. They are based on the Google Bigtable paper, which I highly recommend reading, beware, it is very technical.



*Source: https://database.guide/what-is-a-column-store-database/
Also check out: https://www.saumitra.me/blog/how-cassandra-stores-data-on-filesystem/
https://www.youtube.com/watch?v=HaEPXoXVf2k

Crash course into DBs, and storage

Whatever the DB kind we use, there are always scalability issues. For most of our application-related services, we read more than we write, so the bottlenecks are read latency/throughput.

A way to solve it is just like with services, by replication. Only with some quirks.

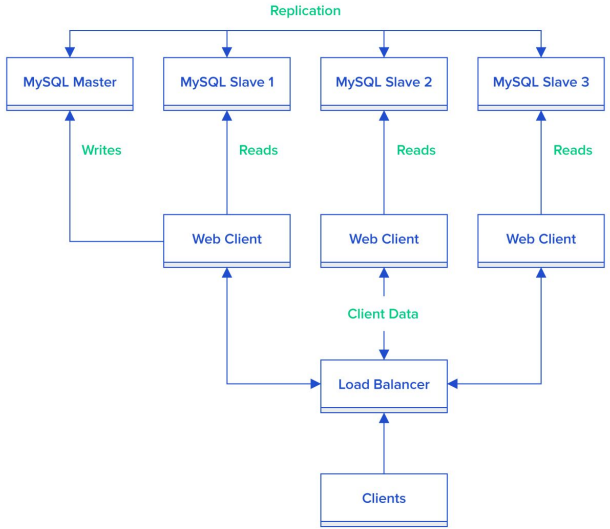
A common architecture is to have read and write databases, where we replicate the read ones, and enable synchronization between write and read copies. To make sure we don’t lose our consistency guarantees, quick synchronous update is possible. Enter so-called master-slave replication. Or leader-follower.

Crash course into DBs, and storage

The principle is fairly simple.

Have all writes go to a single server, then propagate them to the rest. Read only from the slave/followers.

If master/leader crashes, choose another instance as the new leader



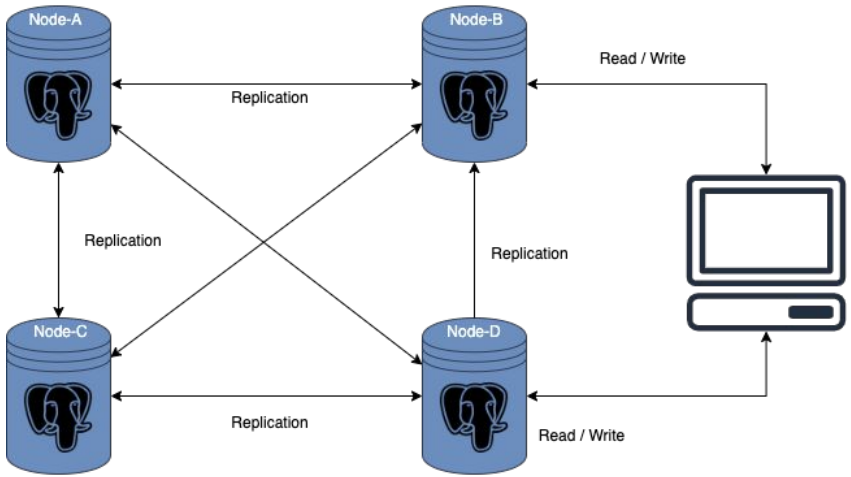
*Source: https://www.toptal.com/mysql/mysql-master-slave-replication-tutorial

Crash course into DBs, and storage

What if now the writes are also plenty?

Multi-master replication to the rescue (kinda).

Multi-master replication is like Marvel's Avengers. They might save the day, in the process severely damaging everything else, leaving you wandering, was it even worth it?



*Source: https://www.percona.com/blog/2020/06/09/multi-master-replication-solutions-for-postgresql/

Crash course into DBs, and storage

If you thought that so far the approach doesn't seem very memory efficient, to keep entire copies of the database running, you are right, it isn't. Meet partitioning. And all it's flavours.

So what's the difference between partitioning and replication. Partitioning still requires multiple instances, just like replication, only now the content of each instance is different, thus we don't waste space.

There are primary 2 types of partitions, horizontal and vertical.

- Horizontal partitioning implies storing subsets of a table, split by row, in different locations.
- Vertical partitioning implies storing same data but split into multiple tables by column. Normalization if a form of vertical sharding.

Crash course into DBs, and storage

Horizontal partitioning is sometimes known as sharding, and is a great enabled for horizontal scaling.

Horizontal partitions can be done at application level and there are different methods to decide how to partition.

Some database servers have partitioning functionality.

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

*Source: https://www.digitalocean.com/community/tutorials/understanding-database-sharding

Crash course into DBs, and storage

When done on a single database, and sometimes even in a distributed environment, partitions help perform queries faster. Why?

Partitions can be done based on input hash, these are so called **hashing** or **key partitions**.

Or on some value (**lookup**) or **range**, like all entries for February have their own table. Or all entries that have the order final price between 15 and 50 USD have a dedicated table.

Beware, sharding also can cause inconsistencies in your system. Check the Pinterest blogpost on how they sharded and scaled their MySQL fleet to get an impression of what can go wrong.

For more info: https://stackoverflow.com/questions/18302773/what-are-horizontal-and-vertical-partitions-in-database-and-what-is-the-differen
https://docs.microsoft.com/en-us/azure/architecture/patterns/sharding
https://medium.com/pinterest-engineering/sharding-pinterest-how-we-scaled-our-mysql-fleet-3f341e96ca6f

NoSQL data modeling

NoSQL, as you might have figured out by now, has a very different data model than classic relational databases. Before we dive into it, we need to understand why NoSQL even exists, and is so different from relational model.

Relational databases were invented in the beginning of 70s, when storage looked like on the right, could store a couple of MB and had a price tag in the order of 100.000s of USD.

Storage was expensive.

*Source: The first *Winchester*: IBM 3340 Storage System, 1973, up to 70MB

NoSQL data modeling

High storage costs were one of the two reasons why people strive for normalization. Second one is to easy the enforcement of functional constraints.

Normalization is reduction of data duplication, basically.

So in order to satisfy users, we need to do many joins.

But what if storage wasn't an issue?

*Source: The first *Winchester*: IBM 3340 Storage System, 1973, up to 70MB

NoSQL data modeling

NoSQL models (recall, there are many of them) are not constrained by storage, because storage is cheap today, and pretty fast. The gist of NoSQL data modeling is:

Derive the structure of your tables from the usage patterns of your application

That is, for each call to the DB, you need to access just one “table”, no joins.

Most of the time it is possible to support multiple use cases with a single table, by the means of composite keys, aggregates, and lack of schema, therefore supporting multiple types of key value pairs in a single table. Check the links below for more info.

For more info: https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/
https://www.youtube.com/watch?v=HaEPXoXVf2k

DBs: concurrency

DBs, be they relational or not, are a priori systems that allow concurrent access. Otherwise, they wouldn't make much sense. But there are different ways to allow safe concurrent access to a DB.

Before we look into how concurrency is handled in DBs, we need to recall the readers-writers problem.

DBs: concurrency

Beside readers-writers problem, throughout history it turned out that it's not so trivial to ensure that transactions do not interfere with one another. Different read and write anomalies were discovered, and to keep DBs safe from them, different transaction isolation levels were proposed.
Read anomalies are:

- **Dirty read**: another transaction may change the value that the first transaction is reading
- **Non-repeatable read and Read skew**: same, but that other transaction has commited
- **Phantom read**: same as above, but the other transaction changes a value that is the subset of read data.

Write anomalies are:

- **Lost updates**: a transaction reads a value, another one writes to it, and the first one writes based on the initial value
- **Write skew**: Two concurrent transactions each determine what they are writing based on reading a data set which overlaps what the other is writing.

DBs: concurrency

DBs are very much like the setup of RW-problem. We have sessions that want to read, and ones that want to write. For this matter, DBs usually have two kinds of locks, so called **shared** or **read lock**, or **exclusive** or **write lock**.

Now that we have locks, how do we ensure data is only written correctly?

We must use either **pessimistic** or **optimistic locking**.

DBs: concurrency

So, let's first dissect the pessimistic locking approach. Basically, apply read/write locks.

To ensure maximum isolation and therefor get rid of most read and write anomalies, we can use two phase locking (2PL). How does it work?

1. We only acquire locks and don't release any
2. We release locks and don't acquire any

DBs: concurrency

In practice? The following sequence uses 2PL:

```
ReadLock(R1), ReadLock(R2), WriteLock(R3),
<perform your actions here>,
WriteUnlock(R3), ReadUnlock(R2), ReadUnlock(R1).
```

This sequence of actions guarantees that no read anomaly will happen, and neither will Lost Updates and Write Skews.

The problem? It's very slow.

For more info: https://begriffs.com/posts/2017-08-01-practical-guide-sql-isolation.html
https://yizhang82.dev/db-isolation-level

DBs: concurrency

Another way to perform concurrency control was inspired by the Universal Scalability Law (recall it?) and does not use locking at all or sparingly. Enter **optimistic locking** or optimistic concurrency control.

Let's run through an example of optimistic locking:

- We want to read R1 and R2, and write into R3
- We make a copy of them all (a snapshot), compute R3
- We check whenever values we wanted to read and write didn't change
- If they didn't, we commit our new changes
- If they did, we abort the transaction and might try again

This method works very well when there are many more reads than writes, or the write contention is small. The downside, we may have some anomalies, usually Write Skews.

DBs: concurrency

Always pick your transaction isolation level depending on your use cases.
Most of them don’t need serialization.

The less isolate the transaction, the more performant your DB will be, in terms of write throughput and latency.

Keywords (Good to know)

RSocket, Thrift, Connection Multiplexing, Hypertext Application Language
Event Sourcing (DDD), Anti-corruption layer, CQRS
Semantic Web, RDF, Consistent hashing, Hi/Lo algorithm

DBs: use cases

When it comes to use cases, two major ones emerge, transactional (OLTP) and analytical (OLAP). OLTP, or online transactional processing, is the scenario that you're most acquainted with, that is, having an operation database where your application writes/reads data to satisfy user necessities defined by the functions of the system/service.

OLAP, or online analytical processing, is a different beast. OLAP systems usually store historical data and use it to answer ad-hoc queries from analysts and business. OLAP systems also keep the data in a denormalized form, for easier querying and reporting.

Most databases, either SQL or NoSQL are primarily focused on OLTP scenarios, but SQL-capable systems (not-necessary relational databases) are a better fit for OLAP scenarios.

DBs: OLAP systems

OLAP systems pretty much break all the rules that you know about the design of a database.

Usually when we're talking about an analytics database we are talking about a **data warehouse**. Data warehouses (DWH) are systems of denormalized databases that hold historical data in a **relational database**. DWH use special schemas, most commonly star and snowflake.

A star schema is used to model business processes as **fact** and **dimension tables**. Example:

```
fact_sales(store_id, product_id, date_id, units_sold)
                |          |          |_ dim_date(id, day_of_week, is_holiday, day, month, year)
                |          |_ dim_product(id, name, brand, sku, price)
                |_ dim_store(id, country, city, address)
```

DBs: OLAP systems

Fact tables represent business processes, like sales, flights, bids, while dimension tables contain information about some specific aspect of the fact table, like time, people, products, places and so on.

Dimensions can have other dimensions, then we have a snowflake schema.

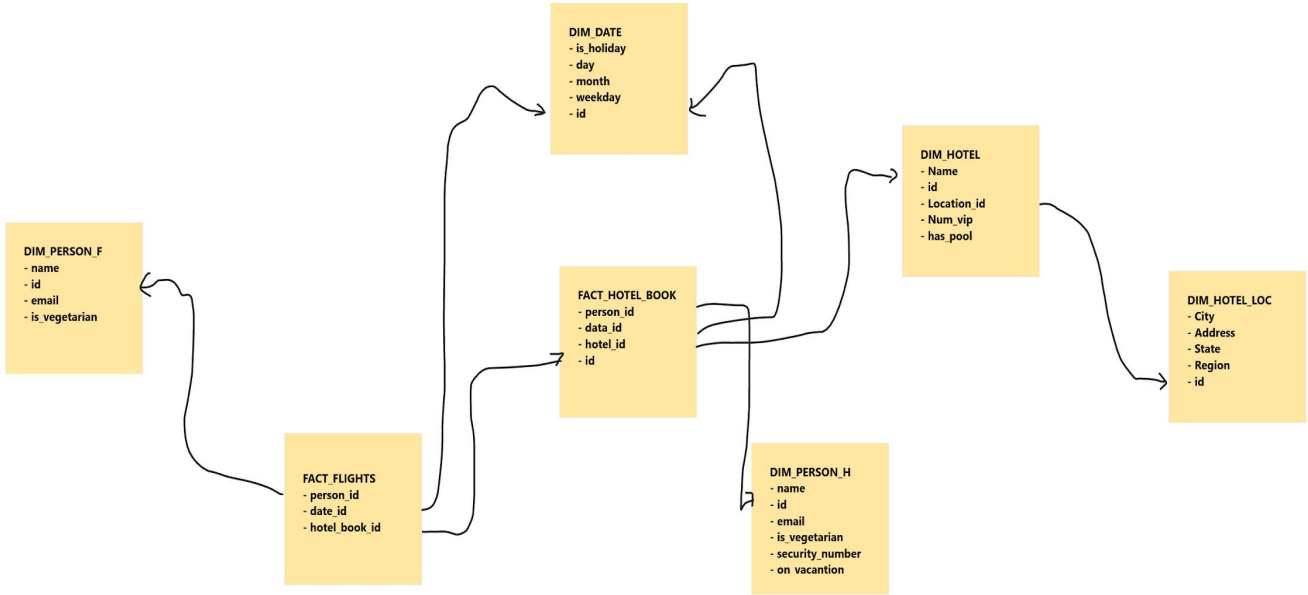
If we have multiple related business processes, like flights, hotel registrations and car rentals, we can say then that we have a fact table constellation.

For a definitive guide, check *R. Kimball, M. Ross, The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*

DBs: OLAP systems

DWHs, like the snowflake schema with fact table constellation, usually are not populated right away with information from the operational database of the application, but through some periodic batch job.

Still, there are examples of Real-Time systems.



DBs: OLAP systems

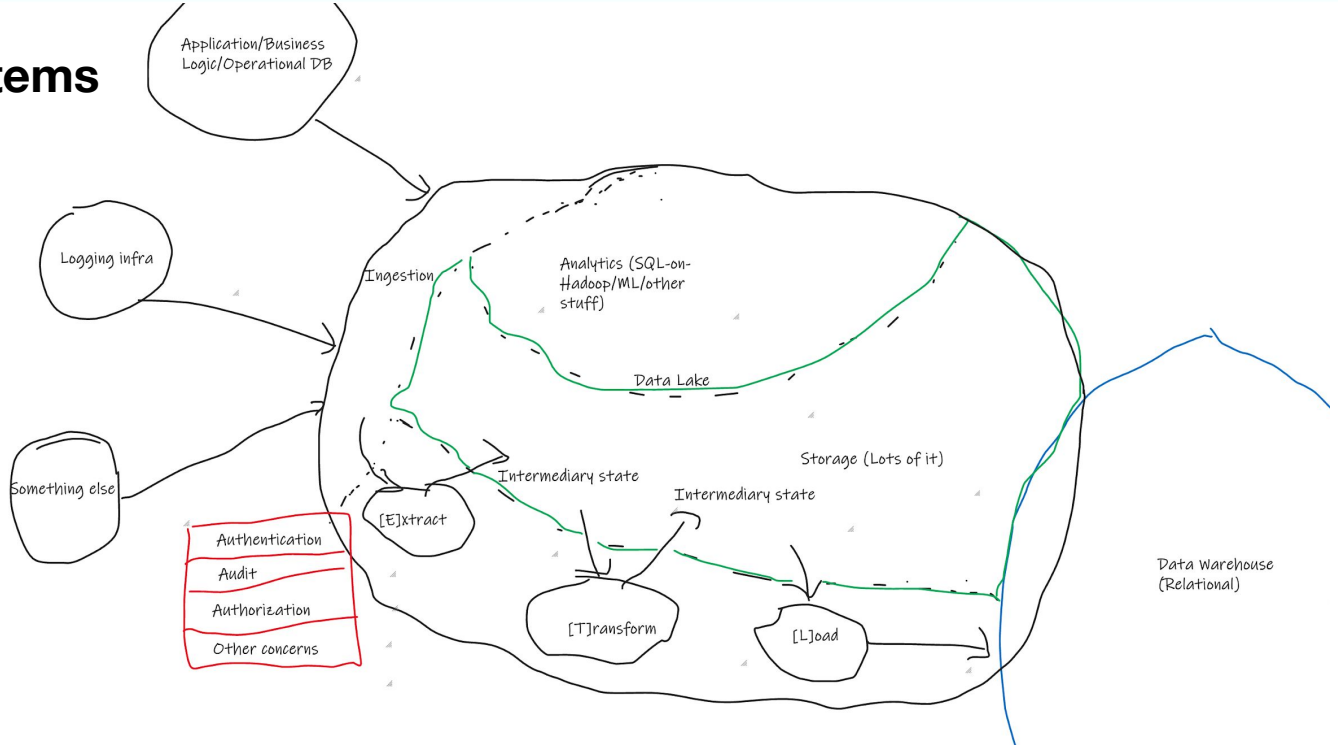
DWHs are quite inflexible, and in the world of Big Data, where data is varied and fast moving, arranging it into a relational database schema can prove to be difficult. That's where Data Lakes come in.

A data lake is a storage and analytics system that's primary purpose is to **ingest** big volumes of both un/semi-structured data, like logs, sensor readings, json/xml documents, and allow for ETL and analytics use cases on it.

DBs: OLAP systems

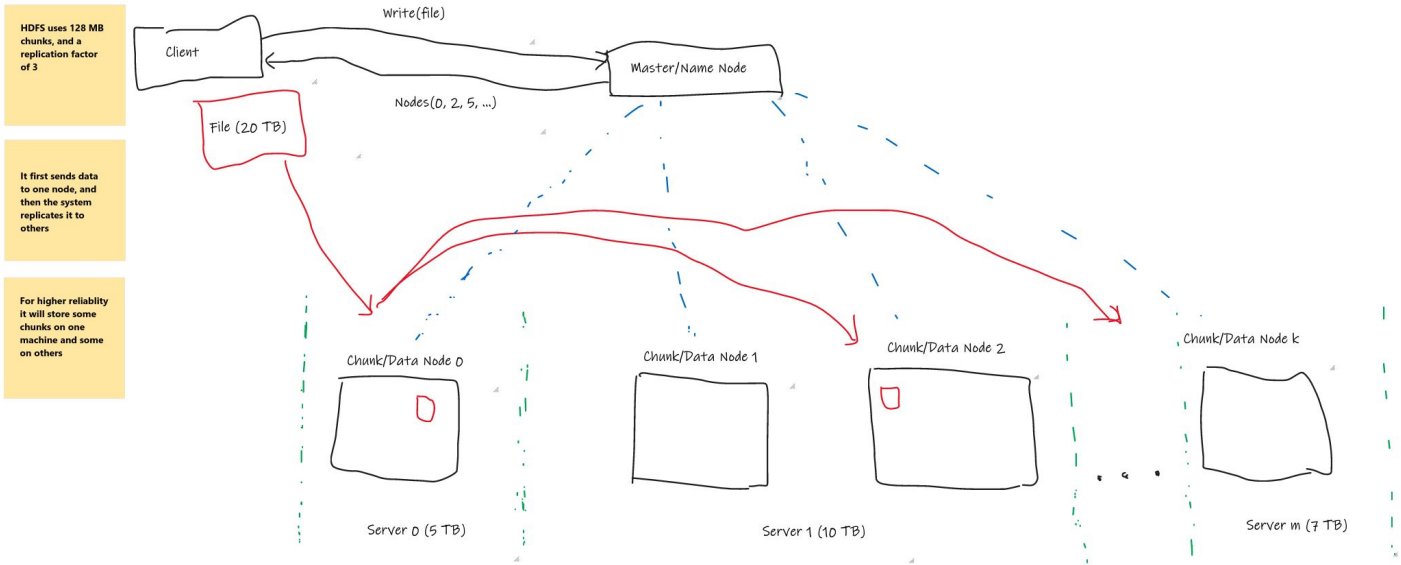
Data Lakes are huge structures, often criticised, still useful.

Storage requirements are usually satisfied with distributed filesystems like Hadoop FS (HDFS) or more recently tools like Amazon's AWS S3.



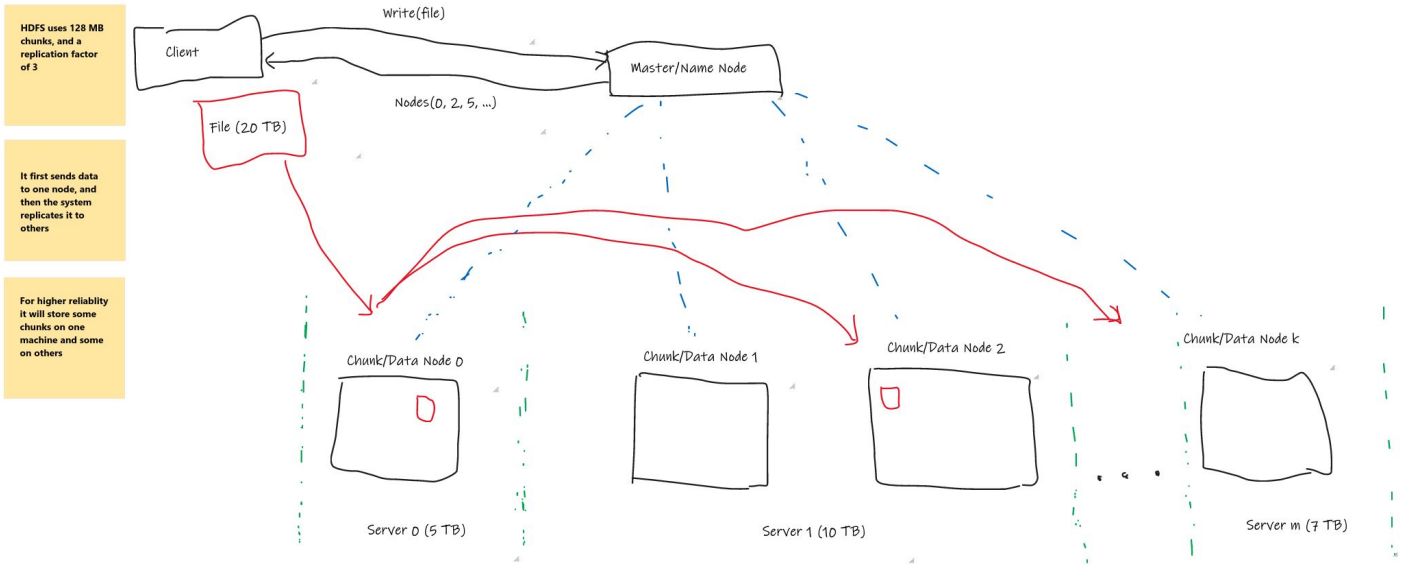
Distributed filesystems: Hadoop filesystem (HDFS)

HDFS was inspired by Google File System, which was primarily designed to be able to reliably keep TBs (even PBs) of data in a distributed manner on commodity hardware that can fail frequently.



Distributed filesystems: Hadoop filesystem (HDFS)

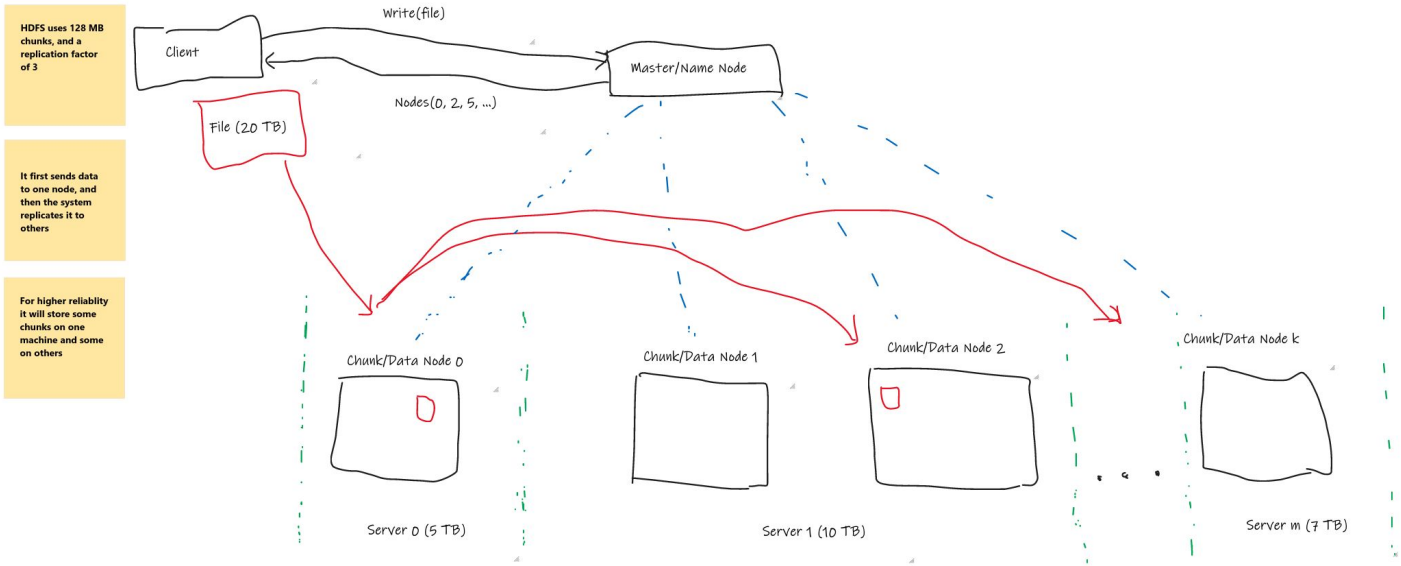
HDFS uses many neat tricks to ensure maximum fault-tolerance. For example rack awareness, which allows for data replicas to be kept on different physical servers.



Distributed filesystems: Hadoop filesystem (HDFS)

A common issue of HDFS was the risk of the NameNode failure and its dire consequences.

As a result, using active-passive failover with tools like Zookeeper to keep masters in sync were used to achieve HA.



Hadoop and MapReduce

MapReduce, a very widely used term at the dawn of Big Data, is a framework and distributed computing paradigm that fit very nicely with HDFS.

MapReduce consists of 3 phases: mapping, shuffling and reducing.

In order to understand MapReduce (MR), you need to understand its main assumptions. MR's primary assumption is that while working with huge datasets (TBs, or PBs) it is foolish to move data to the processing nodes, and therefore the decision was made to move data the least, and instead pass functions/operations to the DataNodes.

Also, given the shaky hardware used by HDFS, MapReduce opts for frequent disk writes of intermediary states of processing, thus ensuring that computations can be more-or-less resumed in case of a node failure.

Hadoop and MapReduce

MapReduce consists of 3 stages:

- Mapping - application of a function/operation to some data
- Shuffling - assembling data and sending via network it to the Reducer nodes for aggregation
- Reducing - grouping data and aggregating it

There are many ways how to tune this process for a given problem, but this is out of scope for this course.

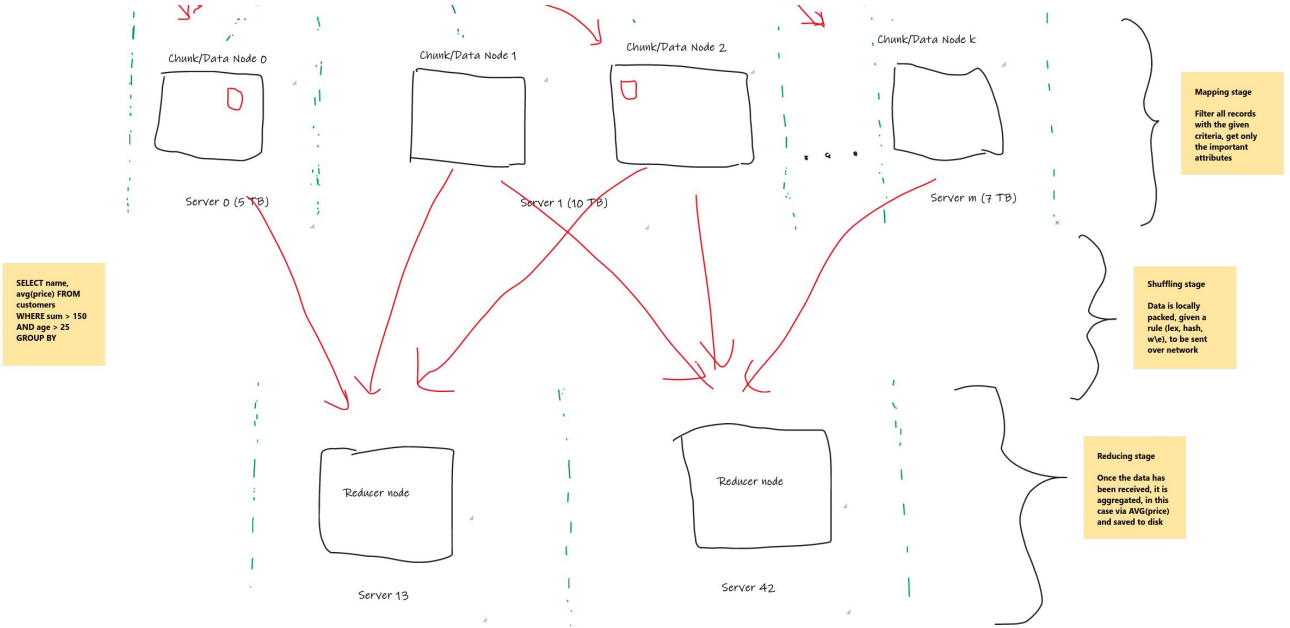
MR only allows DAGs, with no loops, so in order to support iterative computations (ML) it does multiple passes over the data.

For more info on MR patterns and algs: https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/

Hadoop and MapReduce

Another, simpler, example of MapReduce is word count.

Texts |> split()
|> **map(x -> (x, 1), words)**
=via net=> |>
groupBy(word) |> **sum**



Where's Hadoop, there are joins...

Remember that hadoop/MR is used to process big amounts of data, also it sometimes it can generate reports based on the processing done. But reports means **joins**. Remember what a join is is, from your DB course?

First let's talk about classic join algorithms before diving into their distributed counterparts. There are 3 of them (most notable):

- Nested loop join
- Hash join
- Sort-merge join

Where's Hadoop, there are joins...

Nested loop join is fairly simple, and frankly not so efficient, but useful in scenarios where the final result set is not that big, in the order of thousands of elements.

```
for tuple r in RelationR:
    for tuple s in RelationS:
        if r and s satisfy the join condition:
            yield tuple <r,s>
```

As an optimization, making the algorithm aware of the block sizes for each table will increase the processing speed by optimizing for CPU cache.

Where's Hadoop, there are joins...

Next on the list is the hash join, which is considered the least efficient, but useful when only a subset of join keys exist on both (big) tables, thus optimizing for memory use.

```
multimap: Map<Key, List<Values>>
for tuple r in RelationR:
    multimap[r[key1]].append(r)

for tuple s in RelationS:
    for tuple r in multimap[s[key2]]:
        yield tuple <s, r>
```

For implementations of hash join in different langs see: https://rosettacode.org/wiki/Hash_join

Where's Hadoop, there are joins...

Finally, the most efficient join algorithm of all 3, sort-merge join. Still, it shouldn't be thought of as panacea, it is not the best solution when TK

Sort-merge join may be expensive if the data is not ordered in the tables, and as a corollary, it can be very fast if it is stored already sorted.

Assuming the tables are already sorted based on their join keys, we check whenever the current key in one list equals the current key in the other list. If yes, we compute the cartesian product. If one key is greater than the other, we advance the other list and either drop or pad with null the tuple from the first list.

Where's Hadoop, there are joins...

And now for the distributed joins. Hadoop usually has 2 types of joins + variations.

- Reduce-side joins, which are similar to sort-merge joins, and useful for big data volumes.
- Map-side joins, which are similar to hash joins and preferable when one of the tables is small enough to be placed on all used mapper nodes.

A hybrid approach is to do map-side filtering of data, and send it to reduce-nodes for joining, this way reducing the bandwidth requirements.

For pseudocode for both types of joins see: https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/

If Hadoop is too slow for you...

… pick **Apache Spark**. Apache Spark is a big data processing framework, considerably more efficient than Hadoop/MR, but needs more expensive hardware to shine. It comes with tools for ML, graph analytics, SQL-on-hadoop, and streaming analytics support.

So, how does it compare with Hadoop, concept-wise?
First of all, Apache Spark is using **in-memory computing**, diluting a bit the assumption that the nodes are unreliable. In other words, whenever Hadoop writes intermediary results to disk, Spark just keeps them, that is persists, in memory.

If Hadoop is too slow for you - Apache Spark is the way

As a side effect of the Spark behavior, it is possible to reuse intermediary results fairly quickly, allowing for efficient ML and graph processing on multi-node clusters.

The core abstraction of Apache Spark is an **RDD** - resilient distributed dataset.

RDDs are immutable partitions of a dataset placed in a Spark cluster and accessible via a high-level API. Note that RDDs are kept by default in memory and are **not replicated**. So how does it stays fault-tolerant and “resilient”?

If Hadoop is too slow for you - Apache Spark is the way

In case that a partition is corrupted, or simply the underlying server has crashed, Spark finds out and **recreates/reconstructs** that partition. But how??

I just said that RDDs are also immutable. In order to be able to recreate the RDD partition, a **transformation lineage** is kept in memory too. Apache Spark is lazy evaluated, and every transformation applied actually just adds to the DAG. Only when one asks for results it will execute the DAG. This very DAG is used to deduce the lineage.

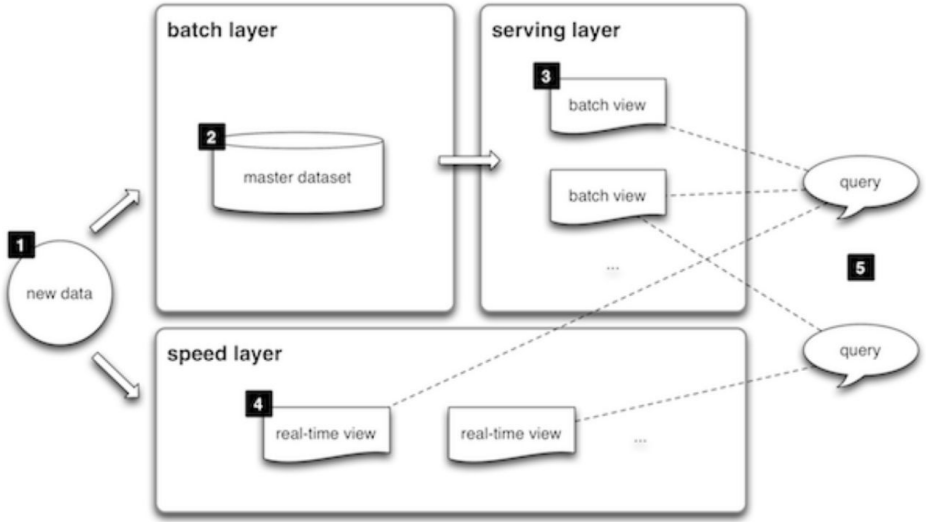
Putting it all together

After defining our tools and being able to process TBs, of data in an interactive fashion, we are still thirsty for quicker insights. In-memory analytics on historic data just doesn't cut it.

Enter real-time analytics. And we will focus only on Lambda and subsequently Kappa architectures. By no means this is a complete discussion on the topic of real-time analytics and streaming data processing.

Putting it all together

First **Lambda Architecture**. Made up of 4 main layers: Ingestion, Batch, Speed and Serving, it combines real-time, simple and sometimes even with a bit of error analytics, and slow but accurate analytics on historical data, in a single architecture.

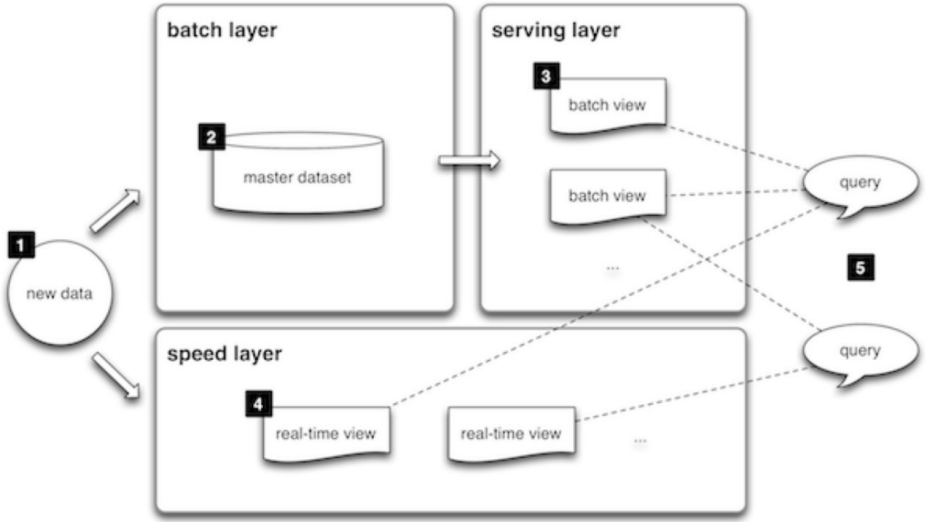


1. All **data** entering the system is dispatched to both the batch layer and the speed layer for processing.
2. The **batch layer** has two functions: (i) managing the master dataset (an immutable, append-only set of raw data), and (ii) to pre-compute the batch views.
3. The **serving layer** indexes the batch views so that they can be queried in low-latency, ad-hoc way.
4. The **speed layer** compensates for the high latency of updates to the serving layer and deals with recent data only.
5. Any incoming **query** can be answered by merging results from batch views and real-time views.

*Source: http://lambda-architecture.net/

Putting it all together

Serving layer is usually a storage that keeps the results of either, or both, of the previous layers and allows for ad-hoc queries and subsequently APIs, Dashboards or spreadsheet tools, to name a few.



1. All **data** entering the system is dispatched to both the batch layer and the speed layer for processing.
2. The **batch layer** has two functions: (i) managing the master dataset (an immutable, append-only set of raw data), and (ii) to pre-compute the batch views.
3. The **serving layer** indexes the batch views so that they can be queried in low-latency, ad-hoc way.
4. The **speed layer** compensates for the high latency of updates to the serving layer and deals with recent data only.
5. Any incoming **query** can be answered by merging results from batch views and real-time views.

*Source: http://lambda-architecture.net/
Also check: https://www.jamesserra.com/archive/2016/08/what-is-the-lambda-architecture/

Putting it all together

There's an issue with the Lambda architecture. It has a lot of redundancy. Both speed and batch layers in principle do the same thing, with minor differences, and when requirements change, we need to update both codebases.

This is how we arrive at Kappa architecture, which basically tells us that *everything is a stream**.

Kappa architecture ditches the batch layer and instead uses windowing, aggregates and rollup strategies to obtain that historical data from an event stream.

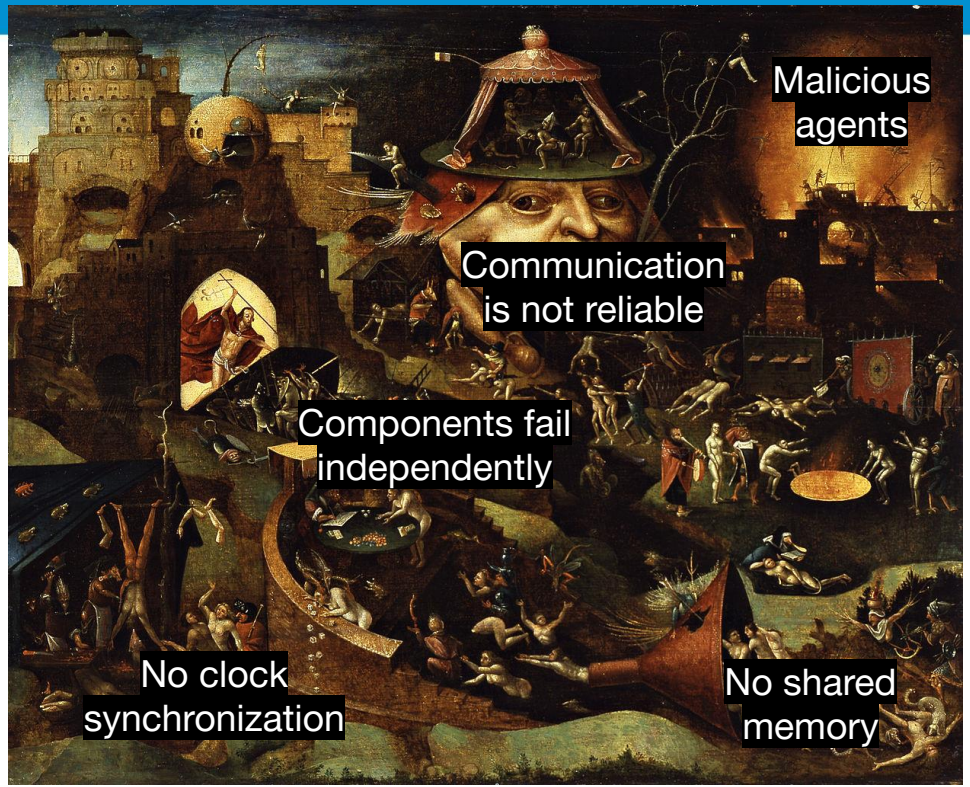
Among the driving forces of Kappa architecture adoption is Apache Kafka, because of its distributed, event-first and persistent nature.

*Check out: https://www.youtube.com/watch?v=fU9hR3kiOK0
and https://jonboulineau.me/blog/architecture/kappa-architecture

Act 2: The (really) hard things

Alexandru Burlacu

Autumn 2020



*Painting: *Christ in Limbo*, a follower of Hieronymous Bosch

Let’s now touch some more fundamental issues.

1. Geographic service distribution
2. Fraud detection systems
3. Tracing/debugging microservices
4. Recovering after some system-wide crash

What do these have in common?

Time. And all the torments that come with it.*

*Check out: https://www.youtube.com/watch?v=-5wpm-gesOY

Time in distributed systems

Imagine a system distributed across the globe, a service like a bank or video streaming. How do you order events like error logs, financial transactions or similar?

Using local time is a very bad idea, I sure hope I don't need to tell you this. Neither is UNIX timestamp a good alternative. So what should you do?

Well, it's complicated. Ofc it is.

But let's first deal with the **data-time format**. Usually, UTC (Coordinated Universal Time) is recommended, but even this is sometimes a bad choice. Depending on what you want to achieve in your system, UTC might have some issues for you, for example the fact that it doesn't correspond to the astronomic time. Or that 23:59:60 is a valid UTC time.

Time in distributed systems

If your systems are sensitive to the time, you might want to use NTP (Network Time Protocol) to synchronize it between your sites, but even it has some issues.

If you are running **very** time-sensitive services or experiments, PTP (Precision Time Protocol) might be a solution for you.

Sometimes to cope with the time-issues, interval timestamps, rather than point timestamps, can be used, but even these are not perfect.

So, what if you don't really care for the real/actual time, but only need a way to order events? Enter **logical clocks**.

*Check out: https://people.cs.aau.dk/~bnielsen/DS-E08/material/clock.pdf

Time in distributed systems

For many systems we don't care as much about the actual time when events occurred, but rather only which events preceded which, in other words their **causal order**.

Recall the term partial order.

Logical clocks are all about that. First proposed by Leslie Lamport, the so-called Lamport timestamps are a way to partially order events in a distributed system, using a fairly simple algorithm.

Time in distributed systems

The algorithm looks like this:

```
P1: # <event is known to be happen now>
P1: time += 1 // A process increments its counter before each event in that process
P1: # <event happens>
P1: send(message, time) // When a process sends a message, it includes its counter value with the message

P2: message, time_stamp = receive()
P2: time = max(time_stamp, time) + 1 // On receiving a message, the counter of the recipient is updated, if necessary, to the greater of its current counter and the timestamp in the received message. The counter is then incremented by 1 before the message is considered received.
```

Time in distributed systems

Given the `->` meaning `happens-before`, he algorithm assumes:

iff `EventA -> EventB => LT(EventA) < LT(EventB)`

The inverse relation **does not hold**.
However, such properties as:

- if `LT(EventA) < LT(EventB)` then either `EventA` might happened-before `EventB` or be concurrent with EventB, but certainly didn't happen after `EventB` in any causal way.
- if `LT(EventA) !< LT(EventB)` then `EventA` didn't happen-before `EventB`

For Lamport timestamps it is possible to obtain total ordering by specifing a conflict resolution strategy, like wall clock, or process pid.

Time in distributed systems

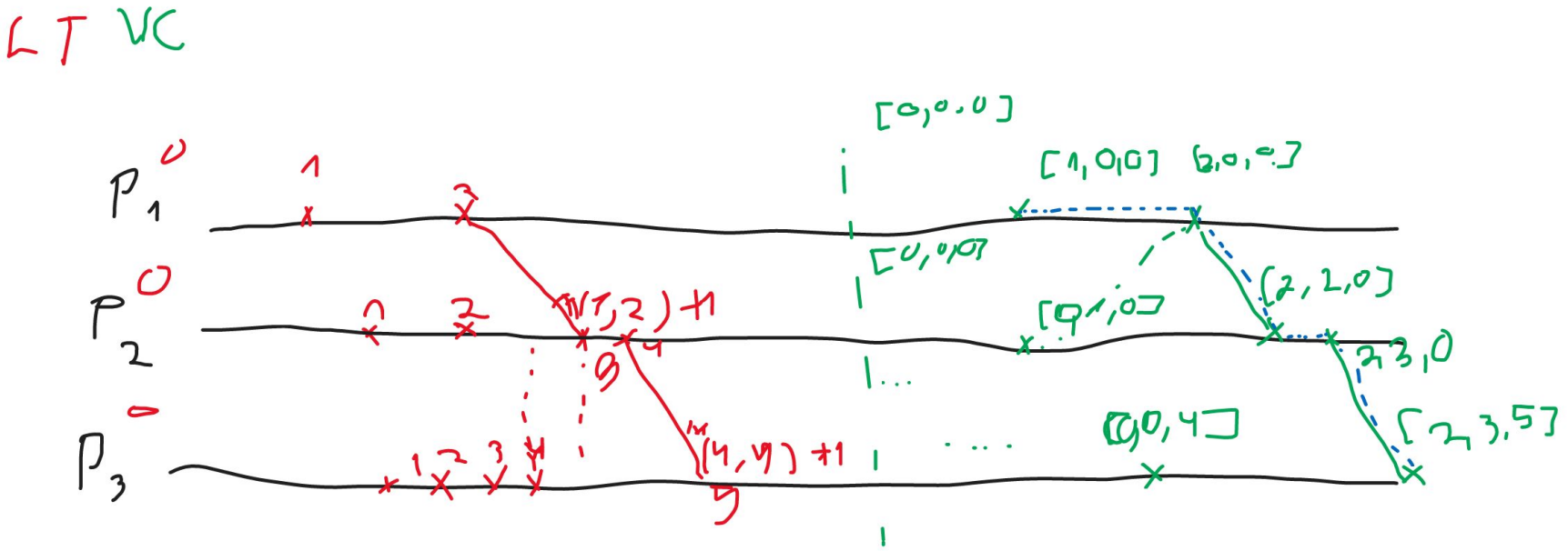
Now for more versatile algorithm, **vector clocks**. A vector clock is an extension of the Lamport timestamp, basically it's a vector of these timestamps of the size N, where N is the number of nodes in a distributed system.

Each process increments only its local clock, but sends the whole vector when interacting with other processes. Upon receiving, the element-wise max is applied to the vector.

Vector clocks are more useful than Lamport timestamps because:
if `EventA -> EventB => VC(EventA) < VC(EventB)`
But now the inverse relation **does hold**.
if `VC(EventA) < VC(EventB) => EventA -> EventB`

*Check out: https://riak.com/why-vector-clocks-are-hard/

Time in distributed systems




strangeloop
Sept 28-30, 2017
thestrangeloop.com
other
timekeeping
mechanisms

Consistency in distribute system

When it comes to databases, we talked about isolation. But all of it was only on a single machine. What if we had a distributed database? **Challenges…**

The good thing about having only one local database is that it usually is very consistent. If I commited a transaction, you can see it right after this. Not so much if we have multiple databases working together.

In principle we could obtain the same behavior, but two things must be taken into account.

- Communication cost/time
- Possible network issues, even partitions

Consistency in distribute system

Before we dive any deeper, what's **consistency**?
When talking about distributed data(bases|stores|w/e) we need to understand that a consistent system will be the one which will have the same data across all nodes.

Now, depending on the concrete system, we can have this happen either "instantly" or after some time.

Most of the time, we want our data to be consistent, at least we think so...

Consistency in distribute system

We want our systems:

- to be as available as possible, that is, when queried, always responding
- to be consistent
- and finally, to resist network failures/partition events, as in nodes are up but can't communicate

Only issue, in real world, you can't have all 3.

Enter CAP theorem.

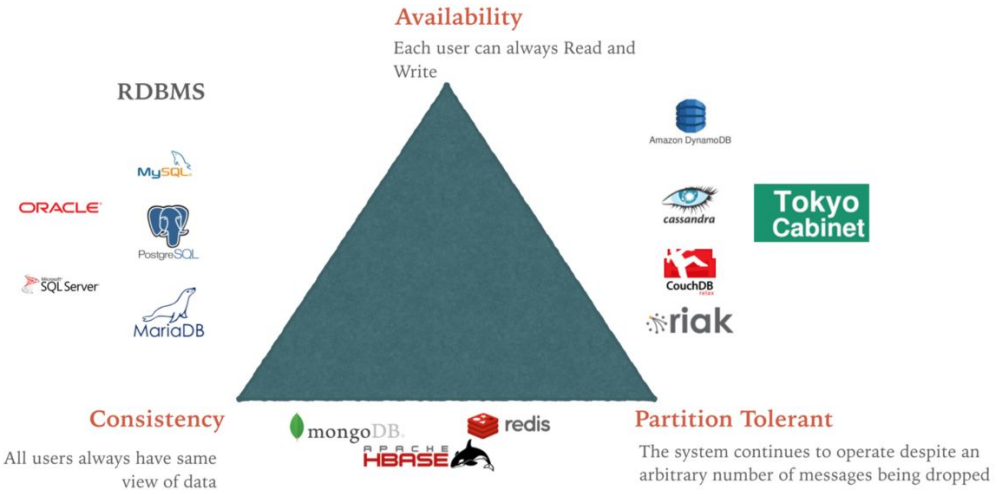
Consistency in distribute system

Enter CAP theorem.

Basically it says, out of Consistency, Availability and Partition Tolerance, pick 2. Or actually just one, because in a distributed setting, you can't do without Partition Tolerance.

In practice Consistency is a spectrum, ranging from eventually consistent (EC) to linearizable, aka atomicly consistent.

Seems a bit incomplete.



Check out: https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/

Consistency in distribute system

So, in practice, PACELC theorem is a more relevant heuristic. PACELC theorem is an extension to the CAP theorem.

Basically, in case of network partition (**P**) in a distributed computer system:

- either availability (**A**)
- or consistency (**C**) (as per the CAP theorem)

must be prioritized, but else (**E**), when the system is running normally, the choice is between latency (**L**) and consistency (**C**).

For example, Cassandra is PA/EL but can tune the consistency levels; both HBase and SQL databases are PC/EC; Mongo is PA/EC; and there are even systems that allow for very high levels of consistency while keeping adequate availability metrics.

Consistency in distribute system

Because of numerous critics regarding CAP theorem and its applicability, here's a nugget of wisdom: **Choose consistency over availability*, but don't overdo it**.

Another nugget of wisdom, usually you won't bother about CAP theorem as much. What will you bother about is the trilemma between (1) access patterns flexibility, (2) latency and (3) throughput and scalability of the system.*

*Check out: https://www.alexdebrie.com/posts/choosing-a-database-with-pie/

Consistency in distribute system

Now, a little throwback to NoSQL databases. During their rise, a lot of implementations where claiming that ACID is usually overkill and that they are BASE (Basically Available, Soft State/tunable consistency, Eventually Consistent).

Let's breakdown:

- **Basically Available**: systems focus on high availability and scalability rather than keeping state consistent, via distribution and replication
- **Soft state**: it is programmer's responsibility to tune the consistency levels of the system
- **Eventually consistent**: system's only guarantee about state is that at some point in future it will converge among all nodes, but no upper bound is given.

*Check out: https://www.lifewire.com/abandoning-acid-in-favor-of-base-1019674

Consistency in distribute system

So, by now we know that SQL databases are consistent, even strongly so, and NoSQL databases are eventually consistent, which is a very weak guarantee. But is this the end of the story? Could it only be only 2 choices?

Until now you might have noticed the assertion that consistency is a spectrum.

On the most constraint end there's **strict consistency**, which is rather a formalism than an actually practical consistency level. It assumes impossibility of concurrent writes to the same variable and require writes to be instantaneous.

A more practical level is linearizability, or **atomic consistency**. This is the maximum you can get in a distributed system. The C in CAP theorem. Even some SQL databases don't provide it.*

*Check out: https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html

Consistency in distribute system

A system is linearizable if given that operation X2 started after operation X1 successfully completed, then operation X2 must see the the system in the same state as it was on completion of operation X1, or a newer state. Quite a strong guarantee.

Between linearizable and eventually consistent systems there are also **causally consistent** ones. More recently, some databases, like Mongo, started providing it.

But even this isn't the end of the story. To ensure causal consistency, systems must have the following special consistency levels: Read-your-writes, Consistent-prefix, Session consistency, those in turn extending Monotonic reads and writes. Different distributed databases have different names and forms of causal consistency.

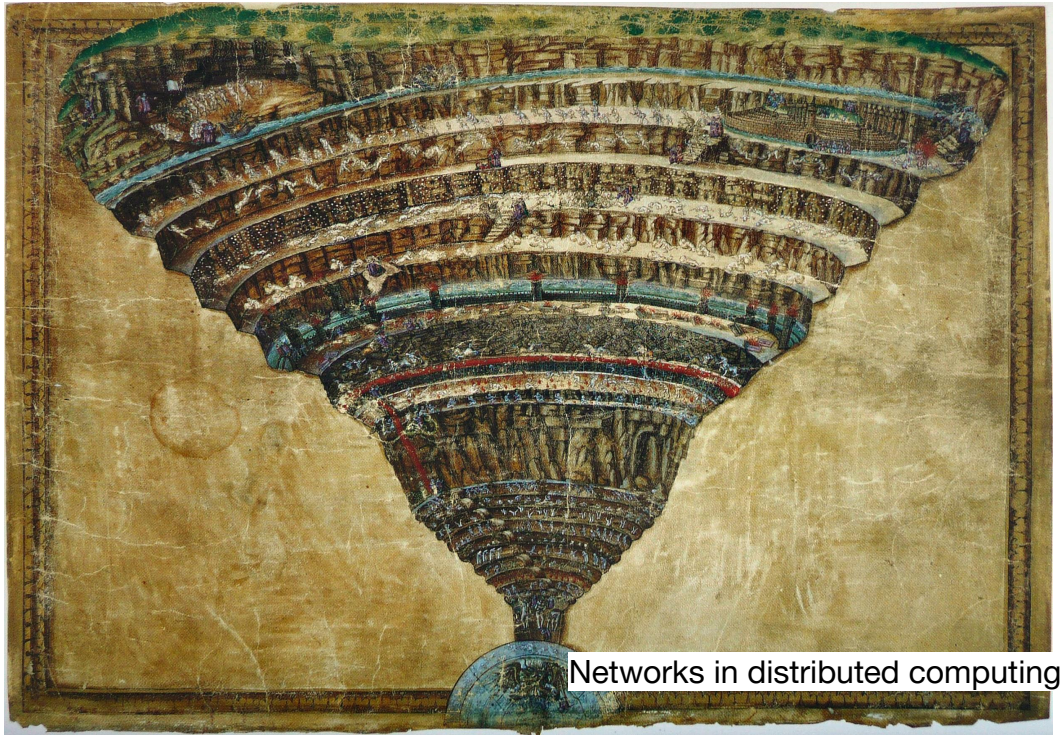
Check out: https://mwhittaker.github.io/consistency_in_distributed_systems/1_baseball.html

Consistency in distribute system

Finally, let's discuss the weakest of them all, **eventual consistency**. Eventual consistency is first of all a liveness guarantee, and does not include safety. In human tongue, eventual consistency guarantees the system is always running, but not that data is correct.

EC systems can guarantee that if no new writes arrive, the data will converge at some point in the future. To ensure convergence, EC needs a way to deal with inconsistent data, either through conflict resolution (reconciliation) or some anti-entropy methods, like an async job that is updating database replicas.
Some of these methods include:

- Riak's vector clocks for reconciliation
- Using the *last-write-wins* for reconciliation
- Cassandra's jobs that synchronize the replicas in a cluster



Networks in distributed computing

Alexandru Burlacu

*Painting: *The Map of Hell, Abyss of Hell* by Sandro Botticelli

Autumn 2020

Treacherous networks and delivery guarantees

Is it possible to ensure that every message sent through the network will always be received, and acknowledged only once?

In other words, is it possible to have **exactly once delivery**?

Treacherous networks and delivery guarantees

Is it possible to ensure that every message sent through the network will always be received, and acknowledged only once?

In other words, is it possible to have **exactly once delivery**?

Nope* but we can try hard and almost always do good.

*Check out: https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/

Treacherous networks and delivery guarantees

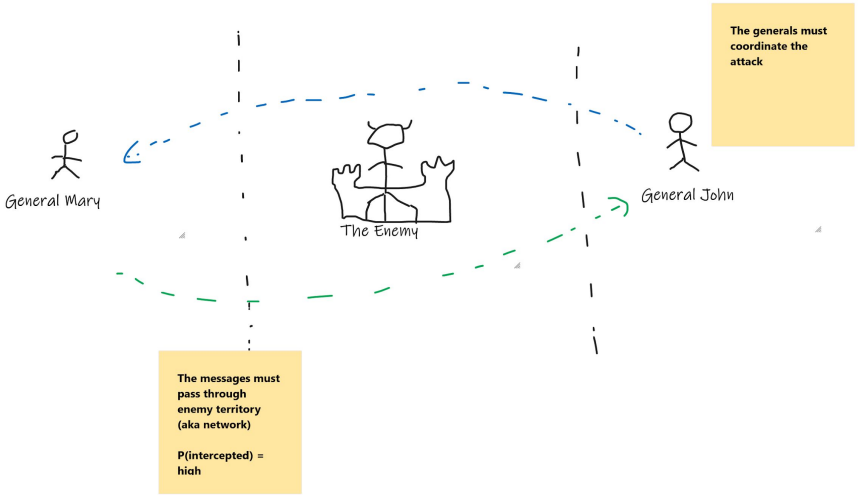
Why exactly once delivery isn't possible?

For this, I must tell you **The Two Generals Problem**.

If sending an ack for the message, how can we ensure it was received and not captured? Another ack?

Using acks is not feasible, because they too are messages and therefore can be lost.

Using timeouts is not good either, why?



Treacherous networks and delivery guarantees

Whenever we're sending packets, messages or streams of data via some network, one important implication of the fallibility of our networks is **will my data be received**?

Depending on the answer you want to this question, there are 3 primary delivery guarantees.

- At most once: fire and forget, no delivery guarantee
- At least once: delivery guaranteed, but might result in duplication
- Exactly once: data is sent, and if delivered, delivery is acknowledged, and if not, data is retransmitted without duplication.

See, the part *without duplication* is the tricky one.

Treacherous networks and delivery guarantees

Exactly once delivery guarantee can’t happen, but exactly once semantic is possible. It actually is correct to call it **essentially once**.

To ensure essentially once delivery, either message deduplication, idempotent processing, or distributed snapshoting must be used.

Still, beware, it is programmer’s responsibility to limit the side effects of message processing, otherwise situations like multiple unwanted database writes, or counter increments are possible.

*Check out: https://www.splunk.com/en_us/blog/it/exactly-once-is-not-exactly-the-same.html

Transactions in distributed systems

Recall what a transaction is.

Local transactions are fairly hard to get right, eliminating any read or write anomalies.
But compared to distributed transactions, it's a piece of cake ©.

We have a number of ways to achieve transactions in a distributed systems scenarios, namely:

- 2 Phase Commits (not to be confused with 2 Phase Locking) for quick operations and low throughput requirements,
- usage of distributed locks or leases that in a way are similar to 2 Phase Commit systems, and finally...
- … Compensating Transactions aka Sagas for long running processes, with relaxed consistency and high throughput requirements.

Transactions in distributed systems

Let's start with distributed locking, the most intuitive one of the three. Just like with classic locking, we can do it both in an optimistic and a pessimistic manner. For optimistic concurrency control in such scenarios ETags are a good proxy to understand the mechanics.

Let's discuss the pessimistic variant.

Imagine, metaphorically, that each service is a "thread" and wants to acquire a "lock" on some resource, possibly part of another service. This metaphoric lock will be a dedicated service, usually named "Lock Manager".

But what will happen if the server that acquired a lock, dies? Or the network between the 2 is congested?

*Check out: https://dzone.com/articles/everything-i-know-about-distributed-locks

Transactions in distributed systems

The answer to this question is to have some kind of lease, rather than a lock. That is, a mechanism that bounds the duration a server can keep the lock.

But there are issues with this approach too. What if the node doesn't know whenever or not its lease has expired?

In principle some sort of heartbeat can be used to keep the locking manager and node with the lease in sync. But of course there are some issues in here too.

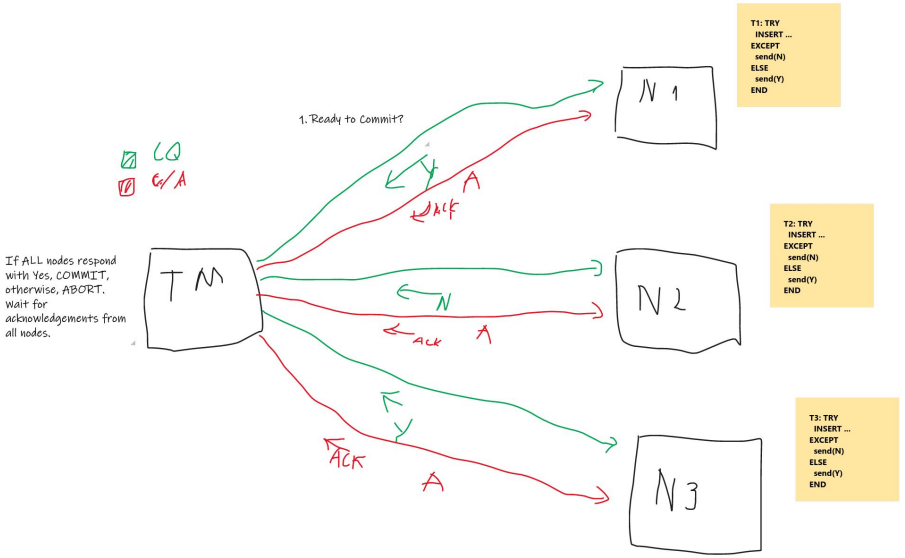
Another approach is to use version, or **fencing tokens**, in a way mimicking optimistic concurrency control schemes.
Popular tools to implement Distributed Locks in a correct way are ZooKeeper, Hazelcast and etcd.

*Check out: https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

Transactions in distributed systems

Another way to do distributed transactions is using a so-called 2 Phase Commit (2PC).

2PC is a scheme of how multiple nodes can agree upon doing a transaction. It works in 2 phases, first the transaction manager node asks the other nodes whenever or not they are ready to do a transaction, and the second phase is actually making the transaction and ensuring that all nodes committed, or aborted, successfully.



*Check out: https://courses.cs.washington.edu/courses/cse452/20sp/slides/2pc.pdf

Transactions in distributed systems

2PC is a blocking protocol, as such it isn't suitable for long running transactions. **Saga**s to the rescue.

A saga, or a long-running process, is a pattern of local transactions that have compensating transactions, in case of failures.

How does it work?
There are 2 primary types

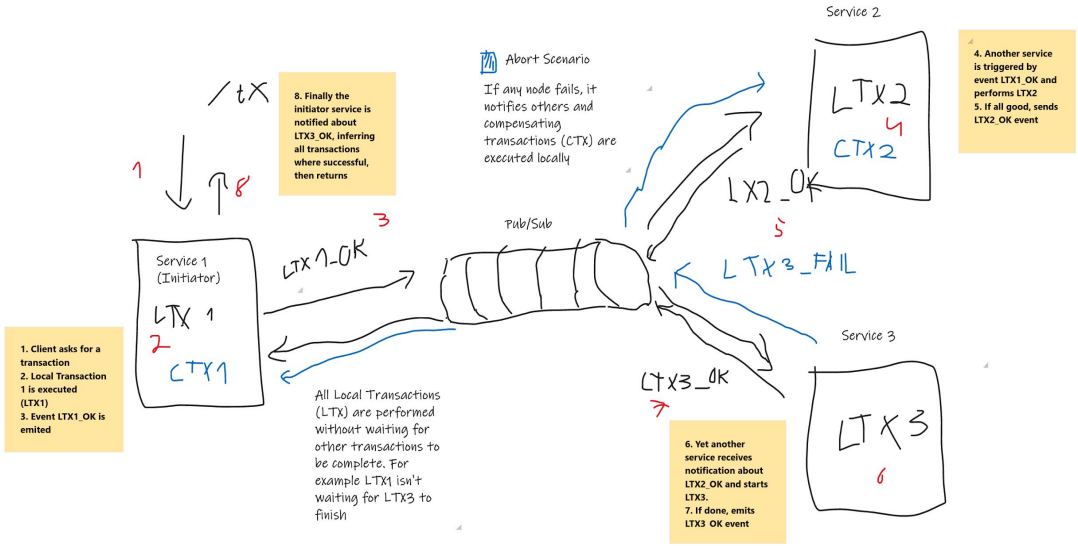
- Orchestrated, via a transaction manager
- and Choreographed, that is nodes communicate between themselves whenever a failure has occurred.

*Check out: https://stackoverflow.com/questions/48906817/2pc-vs-sagas-distributed-transactions

Transactions in distributed systems

Sagas of course have drawbacks, primarily that in order to solve read and write anomalies it is programmer's responsibility to design the system in such a way that either the system can tolerate some level of inconsistency, or that the consistency is enforced at application level.

Still, they achieve higher throughput and are more flexible.



*Check out: https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/ and https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part-2/

Consensus

Systems like transactions are a special case of so called **atomic broadcast** protocols, i.e. we either notify everyone once, or we don't notify at all. Another class of cases requiring strong consistency guarantees are **consensus** protocols.

A consensus is a decision with which everyone agrees with. Think of unanimous voting. Consensus is a necessary condition for many critical components of distributed fault tolerant systems, like configuration management, service discovery, distributed locks, and leader election problems.

So how to achieve consensus?

Consensus

First, the easy-ish part: tools that provide operations with consensus.

Projects like ZooKeeper for the big data ecosystem, etcd for Kubernetes and cloud tools, and also Consul from Hashicorp, all provide tooling based on consensus ideas. Primarily having a key-value or filesystem-like API, these can be used to implement a lot of higher-level distributed and fault-tolerant constructs, like locks, service discovery and other tools.

ZooKeeper uses an algorithm used ZAB* (ZooKeeper atomic broadcast), while etcd and Consul use Raft. Both are based upon the legendary **Paxos** protocol.

*Check out: https://distributedalgorithm.wordpress.com/2015/06/20/architecture-of-zab-zookeeper-atomic-broadcast-protocol

Consensus: it's impossible

Enter the FLP impossibility theorem* - a theoretic and important work, the gist of which is: **you can't guarantee, for all scenarios, consensus if the nodes communicate in an asynchronous fashion over a faulty medium.**

Consensus must have a number of guarantees, namely:

- Termination (a consensus will be reached)
- Agreement/Consistency (all decide on a single value)
- Validity/Integrity (only proposed solutions can be accepted)

Termination is a liveness guarantee, while the other two are a safety guarantee, that is, a guarantee that the algorithm will work correctly.

*Check out: https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/

Consensus: Paxos protocol

Now, enter Paxos protocol. It relaxes the requirement for both safeness and liveness, keeping only the safety guarantee. It was initially proposed by Leslie Lamport and the core idea was to provide a guaranteed consensus protocol, that could be used in practice, and that would overcome the FLP impossibility theorem.

Paxos treats the consensus as an state-machine replication problem, and runs in three stages, **preparation**, **proposal**, and **agreement**.

Paxos can tolerate n failures if 2n+1 machines are used in a cluster. So, in a way, it guarantees liveness for less than n failures. In practice, Paxos, and Paxos-like algorithms are reserved for special cases, because of its slow nature. 3 to 7 node clusters are most popular. More than that is too slow.

*Check out: https://www.cs.princeton.edu/courses/archive/spr11/cos461/docs/lec24-strong.pdf
https://www2.cs.duke.edu/courses/fall07/cps212/consensus.pdf

Consensus: Raft protocol

A protocol similar to Paxos protocol, but easier to grasp, is Raft*.

Paxos is considered to take the optimal number of steps before reaching an agreement, whereas Raft is a bit more sloppy about it, but on the flipside it is considerably easier to implement.

Also, Raft models a Log Replication scenario (see the link to understand what it means). For Paxos to be equivalent with Raft we need to run a so-called Multi-Paxos, that is Paxos for multiple values.

*Check out: http://thesecretlivesofdata.com/raft/

When consistency can be delayed

Sometimes we need to work in fairly large (1k-10k+), dynamic groups, that is, groups that have entities entering and leaving the group. How do we find these members? Or propagate some information to them? Enter **Gossip** or **Epidemic protocols**.

These protocols are a subclass of group membership protocols and are used primarily in P2P networks, like CDNs, BitTorrent networks, and notably in Cassandra clusters for updating state.

The idea is to have nodes know only a subset of the whole group, and randomly push to/pull from them updates. It is provable that such an approach will disseminate the information in $\texttt{ceil}(\log_K(N))$ steps at most. Not strongly consistent, but still some guarantees.

*Check out: https://asafdav2.github.io/2017/swim-protocol/ and https://www.serf.io/docs/internals/simulator.html

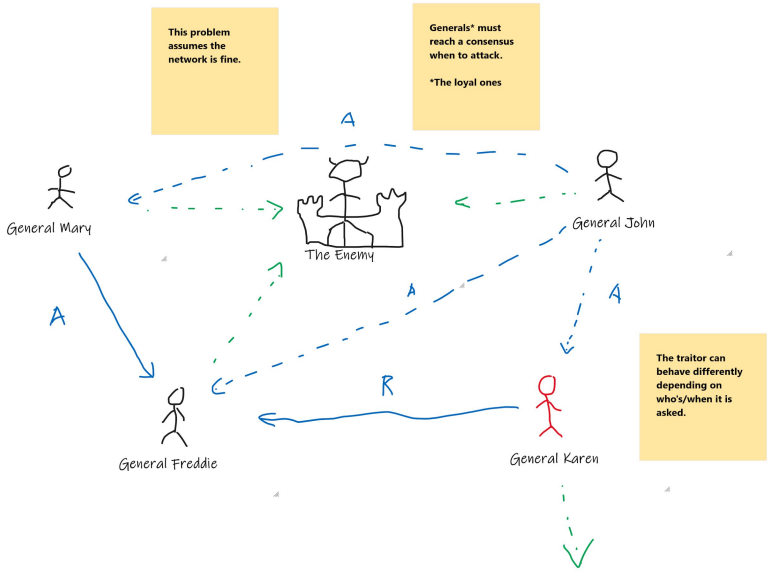
Treacherous agents and consensus

Think this is hard? What if there are more agents/generals, some of which are malicious.

Enter Byzantine Generals Problem, or How to ensure consensus with malicious agents?

Just like simple consensus, Byzantine-fault tolerant systems can't be implemented over asynchroneous channels.

If the channel is synchronous, the cluster requires 3n+1 nodes for n faulty nodes to work properly.

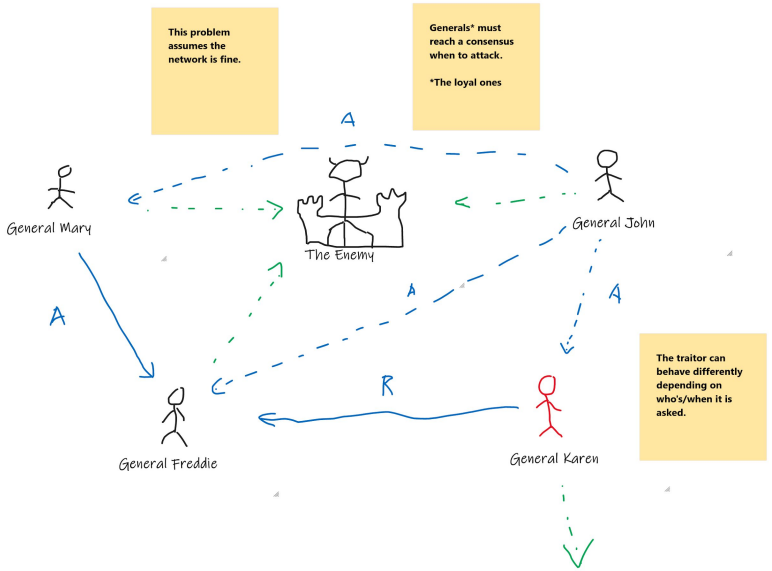


*Check out: https://www.cs.cornell.edu/courses/cs6410/2018fa/slides/18-distributed-systems-byzantine-agreement.pdf

Treacherous agents and consensus

Byzantine fault tolerance is an important computer science topic, with applications in blockchain technology, for example, or mission critical real-time software, were we need to account for occasional faults in sensor readings.

Although a very hard problem, some special solutions do exist, mainly by simulating synchronousness of communication or assuming a not-very-strategic adversary.



*Check out: “Practical Byzantine Fault Tolerance” by Barbara Liskov

Keywords (Good to know)

Edge computing
OLAP Cube, Rollup table, Materialized View
CRDT (Conflict-free Replicated Data Types), Interval Tree Clocks
Chord Protocol, Bloom Filter

Reading list

- https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- https://martinfowler.com/articles/patterns-of-distributed-systems/index.html
- https://lethain.com/introduction-to-architecting-systems-for-scale/
- https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/
- “Enterprise Integration Patterns” - Gregor Hohpe and Bobby Woolf https://www.enterpriseintegrationpatterns.com/docs/EDA.pdf

FACULTATEA
CALCULATOARE, INFORMATICĂ
ȘI MICROELECTRONICĂ

FCIM

TK TK TK
Streaming Algorithms 101

Alexandru Burlacu

Autumn 2020

TK Streaming Algorithms?

http://web.stanford.edu/class/cs368/
http://people.cs.georgetown.edu/jthaler/COSC548.html
https://mapr.com/blog/some-important-streaming-algorithms-you-should-know-about/
https://people.csail.mit.edu/indyk/Rice/lec1.pdf
https://people.cs.umass.edu/~mcgregor/slides/10-jhu1.pdf
https://resources.mpi-inf.mpg.de/departments/d1/teaching/ss14/gitcs/notes3.pdf
http://tutorials.jenkov.com/data-streaming/index.html