blog.pragmaticengineer.com

Distributed architecture concepts I learned while building a large payments system

Gergely Orosz

19-24 minutes

When building a large scale, highly available and distributed system, what architecture concepts do you need to use, in practice? In this post, I am summarizing ones I have found essential to learn and apply when building the payments system that powers Uber. This is a system with a load of up to thousands of requests per second, where critical payments functionality needs to work correctly, even if parts of the system are down.

As background: I joined Uber two years ago as a mobile software engineer with some backend experience. I ended up building the payments functionality in the app - and rewriting the app on the way. Afterwards, I ended up moving into engineering management, heading up the team itself. This meant getting exposed to more backend, as my team is responsible for many of the backend systems that enable payments.

Before working at Uber, I had little to no distributed systems experience. My background is a traditional computer science degree and a decade of full stack software development. However, while I was able to draw boxes and talk tradeoffs, I did not have much understanding or appreciation of distributed concepts like consistency, availability or idempotency.

Is this list that I'll go over complete? Probably not. But it's the stuff that would have made my life easier, had I known them earlier. So let's dive into things like SLAs, consistency, data durability, message persistence, idempotency and some other things I needed to learn on the job.

[Update] Read also the follow-up post for this article: Operating a large, distributed system in a reliable way and I also recommend the book Designing Data-Intensive Applications for more reading.

SLA

With large systems, that process millions of events per day, some things are bound to go wrong. Before diving into planning a system, I have found the most important thing to decide what a system that is "healthy" means. "Healthy" should be something that is *actually* measurable. The common way to measure "healthy" is with SLAs: service level agreements. Some of the most common SLAs I have seen used are:

- **Availability**: the percentage of the time the service is operational. While it is tempting to want to have a system that has 100% availability, achieving this can be really difficult, as well as expensive. Even large and critical systems like the VISA card network, Gmail or internet providers don't have 100% availability - over years, they will be down for seconds, minutes or hours. For many systems, the four nines availability (99.99%, or about 50 minutes downtime per year) is considered high availability. Just getting to this level is quite the work usually, to get to.
- **Accuracy**: is it ok for some of the data in the system to be inaccurate or lost? If so, what percentage is acceptable? For the payments sytems that I worked on, accuracy needed to be 100%, meaning no data was allowed to be lost.
- **Capacity**: what expected load should the system be able to

support? This is usually expressed in requests per second.

- **Latency**: in what time should the system respond? What is the time that 95% of the requests and 99% of the requests should be served? Systems usually have a lot of noisy requests, hence the p95 and p99 latencies are more practical usage in the real world.

  Why did SLAs matter when building a large payments system? We put together a new system, replacing an existing one. To make sure we build the right thing, a system that is "better" than it's predecessor, we used SLAs to define expectations. Availability was one of our top requirements. Once defining the target, we needed to consider the tradeoffs in the architecture to be able to meet this.

Horizontal vs vertical scaling

Assuming the business using a newly built system grows, the load will only increase. At some point, the existing setup will not be able to support more load and more capacity needs to be added. The two most common scaling strategies are vertical or horizontal scaling.

Horizontal scaling is about adding more machines (or nodes) to the system, to increase capacity. Horizontal scaling is the most popular way to scale distributed systems, especially, as adding (virtual) machines to a cluster is often as easy as a click of a button.

Vertical scaling is basically "buying a bigger/stronger machine" - either a (virtual) machine with more cores, more processing, more memory. With distributed systems, vertically scaling is usually less popular as it can be more costly than scaling horizontally. However, some major sites, like Stack Overflow has successfully scaled vertically to meet demand.

Why did the scaling strategy matter when building a large payments system? We decided early on that we would build a system that scales horizontally. While vertical scaling is possible in

some cases, our payments system was already at the projected load that we were pessimistic if a single, super-expensive mainframe could even handle it today, not to mention in the future. We also had engineers on our team who have worked at large payment providers where they tried - and failed - to scale vertically on the largest machines that money could buy at their time.

Consistency

Availability of any system is important. Distributed systems are often built on top of machines that have lower availability. Let's say our goal is to build a system with a 99.999% availability (being down about 5 minutes/year). We are using machines/nodes that have, on average, 99.9% availability (they are down about 8 hours/year). A straightforward way to get our availability number is to add a bunch of these machines/nodes into a cluster. Even if some of the nodes are down, others will be up and the overall availability of the system will be higher, than the availability of the individual components.

Consistency is a key concern in highly available systems. A system is consistent if all nodes see and return the same data, at the same time. Going back to the previous model, where we added a bunch of nodes to achieve higher availability, ensuring that the system stays consistent is not so trivial. To make sure that each node has the same information, they need to send messages to each other, to keep themselves in sync. However, messages sent to each other can fail to deliver, they can get lost and some of the nodes might be unavailable.

Consistency is a concept that I spent the most time understanding and appreciating. There are several consistency models, the most common one used in distributed systems being strong consistency, weak consistency and eventual consistency. The Hackernoon article on eventual vs strong consistency gives a nice and practical

overview of what the tradeoffs between these models are. Typically, the weaker the consistency required, the faster the system can be, but the more likely it will return not the latest set of data.

Why did consistency matter when building a large payments system? Data in the system needed to be consistent. But just how consistent? For some parts of the system, only strongly consistent data would do. For example, knowing if a payment has been initiated is something that needed to be stored in a strongly consistent way. For other parts, that were less mission-critical, eventual consistency is something that was considered as a reasonable tradeoff. A good example is listing recent transactions: these could be implemented with eventual consistency (meaning, the latest transaction might only show up in parts of the system after a while - in return, the operation will be return with lower latency or be less resource intensive).

Newsletter

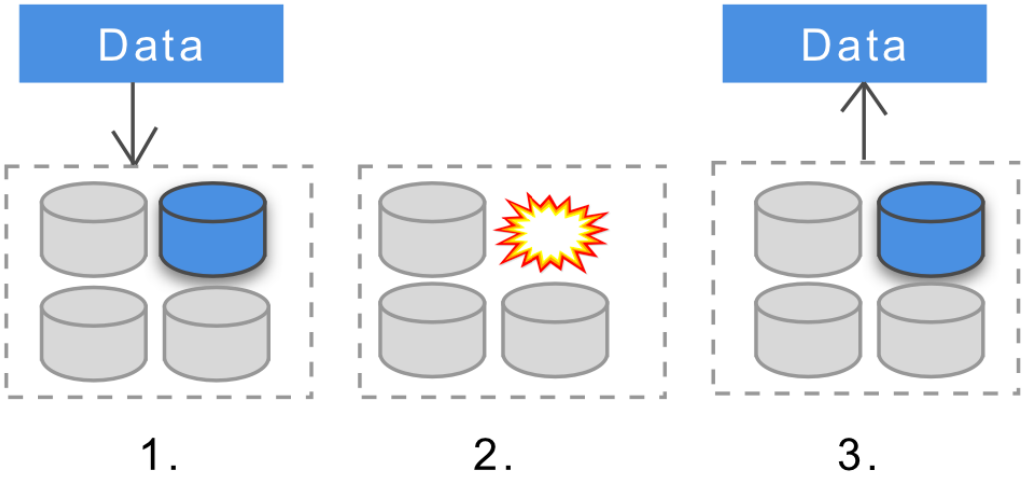
Subscribe to my weekly newsletter for advice, observations and inspiration across the software engineering industry. It's the #1 technology newsletter on Substack with over 120,000 readers.

Data Durability

Durability means that once data is successfully added to a data store, it will be available going forward. This will be the case even if nodes in the system go offline, crash or have their data corrupted.

Different distributed databases have different levels of durability. Some support machine/node level durability, some do this at cluster level and some just don't provide it out of the box. Some form of replication is usually used to increase durability - if the data is stored on multiple nodes, if one or more nodes go down, the data will still be available. Here is a good article on why achieving

durability in distributed systems can be challenging.



Why did data durability matter when building a payments system? For many parts of the system, no data could be lost, given this being something critical, like payments. The distributed data stores we built on needed to support cluster level data durability - so even if instances would crash, completed transactions would persist. These days, most distributed data storage services, like Cassandra, MongoDB, HDFS or Dynamodb all support durability at various levels and can be all configured to provide cluster level durability.

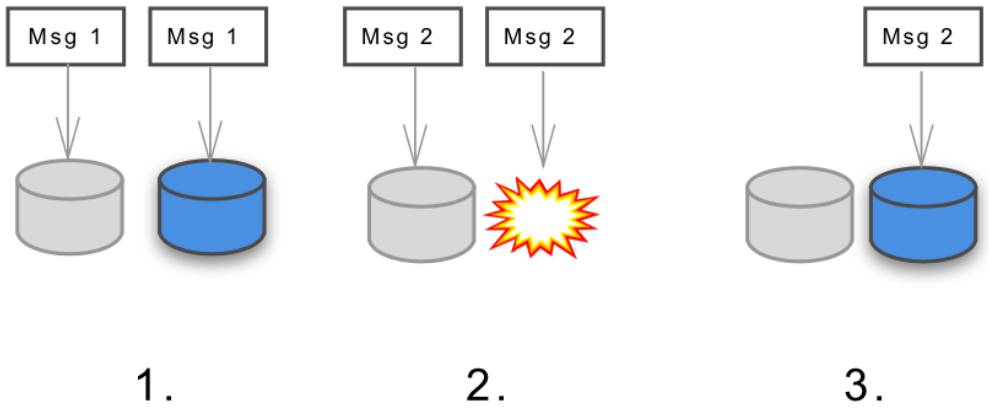
Message Persistence and Durability

Nodes in distributed systems perform computations, store data and send messages to each other. A key characteristic of sending messages is how reliably these messages arrive. For mission-critical systems, there is often a need for having zero messages being lost.

For distributed systems, messaging is usually done by some distributed messaging service, such as RabbitMQ, Kafka or others. These messaging services can support (or be configured to support) different levels of reliability in delivering messages.

Message persistence means that when some failure happens on a

node that is processing the message, the message will still be there to process after the failure is resolved. Message durability is usually used at message queue level. With a durable message queue, if the queue (or node) goes offline when the message is sent, it will still get the message when it comes back online. An good article to read more on this topic is this one.



Why did message persistence and durability matter when building a large payments system? We had messages that could not be afforded to be lost, such as the message that a person has initiated payment for their ride. This meant that the messaging system that we used needed to be lossless: every message needed to be delivered once. However, building a system that delivers each message *exactly* once or one that delivers *at least* once is different complexity. We decided to implement a durable messaging system with at least once delivery and chose a messaging bus, on top of which we would build this (we ended up going with Kafka, setting up a lossless cluster for this case).

Idempotency

With distributed systems, things can go wrong, such as connections could drop midway or requests can time out. Clients will often retry these requests. An idempotent system ensures that no matter, how many times a specific request is executed, the actual execution on

this request only happens once. A good example is making a payment. If a client makes a request to pay, the request is successful, but the client times out, the client could retry this same request. With an idempotent system, the person paying would not get charged twice. With a non-idempotent system, they could.

Designing for idempotent, distributed systems require some sort of distributed locking strategy. This is where some of the earlier distributed system concepts come into play. Let's say we intend to implement idempotency by having optimistic locking in place, to avoid concurrent updates. In order to have optimistic locking, the system needs to be strongly consistent - so that at the time of the operation, we can check if another operation has been initiated, using some sort of versioning.

There are numerous ways to achieve idempotency, depending on the constraints of the system and the type of operation. Designing idempotent approaches is a nice challenge - Ben Nadel writes about different strategies he has used, both with distributed locks or database constraints. When designing distributed systems, idempotency can be one of the easily overlooked parts. I have come across scenarios, where my team was burned by not ensuring correct idempotency for some key operations.

Why did idempotency matter when building a large payments system? Most importantly: to avoid double charges or double refunds. Given that our messaging system has at least once, lossless delivery, we need to assume that all messages might be delivered multiple times and systems need to ensure idempotency. We chose to handle this with versioning and optimistic locking, having the sytems that implement idempotent behaviour use a strongly consistent store as their data source.

Sharding and Quorum

Distributed systems often have to store a lot more data, than a single node can do so. So how does one go about storing a bunch of data on a certain number of machines? The most common technique is using sharding. Data is horizontally partitioned using some sort of hash to assign to a partition. While many distributed databases implement sharding under the hood, sharding is an interesting area to learn more about, especially around resharding. Foursquare had a 17 hour downtime in 2010 due to hitting a sharding edge case, where a nice postmortem was shared on the root causes.

Many distributed systems have data or computations replicated across multiple nodes. In order to make sure that operations are performed in a consistent way, a voting based approach is defined, where a certain number of nodes needs to get the same result, for the operation to be successful. This is called the quorum.

Why did quorum and sharding matter when building the payments system at Uber? Both of these are basic concepts that are pretty commonly used. I personally came across this concept when looking into how we setup Cassandra replication. Cassandra (and other distributed systems) use quorum and local quorum to ensure consistency across clusters. As a funny side effect, on some of our meetings, when enough people are in the room, someone would ask: "Can we start? Do we have a quorum?"

The Actor Model

The usual vocabulary of describing programming practices - things like variables, interfaces, calling methods - all assume single machine systems. When talking about distributed systems, we need to use a different set of approach. A common way of describing these systems is following the actor model, where we think about the code in terms of communication. This model is popular, as it matches the mental model that we would think of, for

example, when describing how people communicate in an organization. Another, also popular way of describing distributed systems is CSP - communicating sequential processes.

The actor model is based on actors sending messages to each other and reacting to them. Each actor can do a limited set of things - create other actors, send messages to others or decide what to do with the next message. With a few simple rules, complex, distributed systems can be described well, which can also repair themselves, after an actor crashes. For a short overview, I recommend the article The actor model in 10 minutes by Brian Storti. Many languages have implemented actor libraries or frameworks. For example, at Uber, we use the Akka toolkit for some of our systems.

Why did the actor model matter when building a large payments system? We were building a system with many engineers, a lot of them having distributed experience. We decided to follow a standard distributed model over coming up with distributed concepts ourselves, potentially reinventing the wheel.

Reactive Architecture

When building large, distributed systems, the goal is usually to make them resilient, elastic and scalable. May this be a payments system or another, high load systems, the patterns in doing this can be similar. People in the industry have been discovering and sharing best practices that work well in these cases - and Reactive Architecture is a popular and widely used pattern in this space.

To get started with Reactive Architecture, I suggest reading the Reactive Manifesto and watching this 12-minute video on the topic.

Why did Reactive Architecture matter when building a large payments system? Akka, the toolkit we used to build much of our new payments system was heavily influenced by reactive

architecture. Many of our engineer's building this system were also familiar of reactive best practices. Following reactive principles - building a responsive, resilient, elastic and message-driven system, thus came quite naturally. Having a model to fall back on and check that progress is on the right track was something that I found helpful and I'll be using this model when building future systems as well.

Closing

I have been lucky enough to participate in the re-building a high-scale, distributed, mission-critical system: the one powering payments at Uber. By working in this environment, I picked up a lot of distributed concepts that I did not have to use previously. I've summarized these in the hope that others find this helpful to start or continue learning about distributed systems.

This write-up has been strictly focused on the planning and architecting of these systems. There is a whole lot of things to be said about building, deploying and migrating between high-load systems as well as operating them reliably. But all of those are topics for another post: Operating a large, distributed system in a reliable way.

For those interested in learning more on distributed systems, I recommend the book Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. This is the most practical book I've found so far. It covers the different distributed concepts - both those that I've gone through and some more - together with real-world examples. The Amazon Builder's Library also has some more specific, in-depth topics on distributed systems.

Note: read the Russian translation of this post, as well as the Japanese translation.
