Distributed System Design

theroyakash

Masters in Computer Science
Indian Institute of Technology, Madras
Chennai, India

Winter of 2022 to Summer of 2023

Contents

Chapter 1

Introduction to Distributed systems

1.1 What is a distributed system?

There is a running joke among the distributed system designers is that the definition of such system is "I can **NOT** get my work done because some computer somewhere has failed that I never knew existed". More formally a distributed system is such a system that is

- running on more than one **Node**
- nodes are interconnected by Computer Networks
- characterized by partial failures

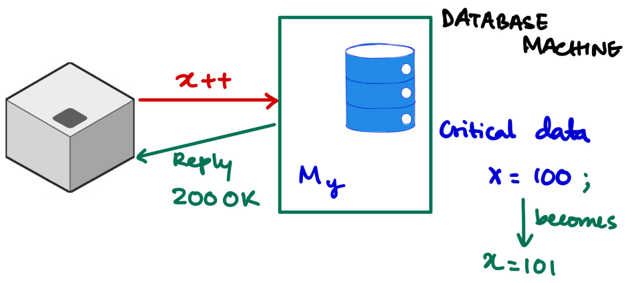
Here a **Node** is a computer or an instance of an application that is running on a computer and *partial failure* means some part of the system is working and some part may not be working.

1.2 Failures, problems in distributed systems

So a distributed system is defined based on the failure of it's components. So there are many types of failure that should be accounted for. Mainly two broad *philosophy* are used to deal with failures in a distributed system.

- **High-Performance Computing** philosophy where we treat partial failures as total failure and start the entire operation from start or from previously saved checkpoints
- **Cloud Computing** philosophy where we treat the partial failures as a component failure and try to mitigate the fallout from the failure by working around it. The main philosophy here is that every thing fails eventually, so make a system that is **fault tolerant**.

Consider a two **Node** system M_X and M_Y is connected via a computer network some distance [1600 KM let's say] apart. Now the machine M_Y has some critical data that M_X needs to get hold of and increment by 1. Think of the failures that can occur during this whole operation.



Things that can *fail* in this system are the following

- The command `x++` never transmits to the Machine M_Y, gets lost during transit,
- M_Y *crashes* during the execution of `x++` so the reply `200 OK` never comes back to machine M_X,
- The command `x++` gets corrupted in transit,
- `x++` executes alright but the reply `200 OK` never comes back to machine M_X,
- `x++` executes alright but the reply `200 OK` get corrupted in transit to machine M_X,
- Value of X got changed while the reply `200 OK` is in transit back to machine M_X
- Both the system is running with lot less CPU power than expected rendering the whole system slow.

Although we have so much problems with distributed systems we still make systems distributed for the following reasons RELIABILITY OF DATA, HIGH THROUGHPUT, FASTER EXECUTIONS. And **Yes** if you are thinking that some of those above problems can be solved with TCP in the transport layer of the network, you are absolutely correct.

1.3 When to use distributed systems?

With so many problems in a distributed systems why would anybody having a sane mind would go for a distributed systems though? The answer is pretty simple, you wouldn't unless you **have to**. When the problem is too big, too heavy to solve even for one machine you will have to solve it with distributed systems. For example let's take the following system:

Your business is an online exam preparation course (for example say JEE or GATE coaching), you have about 100,000 candidates registered with your course and they study notes from your website, take tests and see their evaluation. Let's see how much load this website gets.

- During normal times about 5-8 months before the actual exam website gets about $\approx$ 4000-5000 people in certain hour concurrently watching videos, studying notes and watching solutions, and maybe 500 people are taking short exams on the website's exam portal to prepare.
- During an All India Mock test $\approx$ 75000 - 80000 candidates signs on to the portal to give exam and order of 100s of candidate skips the exam to watch videos.

Now if you see, a decent **AWS** virtual machine server running linux with 8 GB RAM with 4 core CPU and about 200 GB worth of disk space for persistent database would be fine to run during normal times as discussed above. During all india exams we may have to scale the server to 32

GB RAM, and 16 Core CPU to handle all of the candidates. But more or less with one virtual machine it is sufficient for this company to run a multi-crore rupees business online. So they most definitely do not need a distributed system for their business, going distributed is not worth the money and effort to maintain for this case.

If your business

- handles in the order of 100-200 million requests every minutes,
- has 50s, 100s of terabytes of worth of business critical data that must not be lost,
- has millions of requests for data delivery half-way accross the world,
- too big of a service to fail even for a minute in any given time, date, place, season (like YouTube, Amazon, Netflix after a new popular season launch, BookMyShow, Swiggy, Zomato food services, Hotstar during a live cricket match of India vs Pakistan, JEE-GATE like all India Exams)

then you need to use distributed systems and have to deal with all the problems that arises with those system designs. One more thing - do not get carried away with distributed systems so much that you even lose common sense. For example design me a content delivery service that handles large north of 50 Petabytes of data from one existing data center to a new data-center to reach more customers in a new city. Even though you have to deliver a lot of data to this new service center, traditional methods far outweighs any distributed systems you may design for this high volume of content delivery.

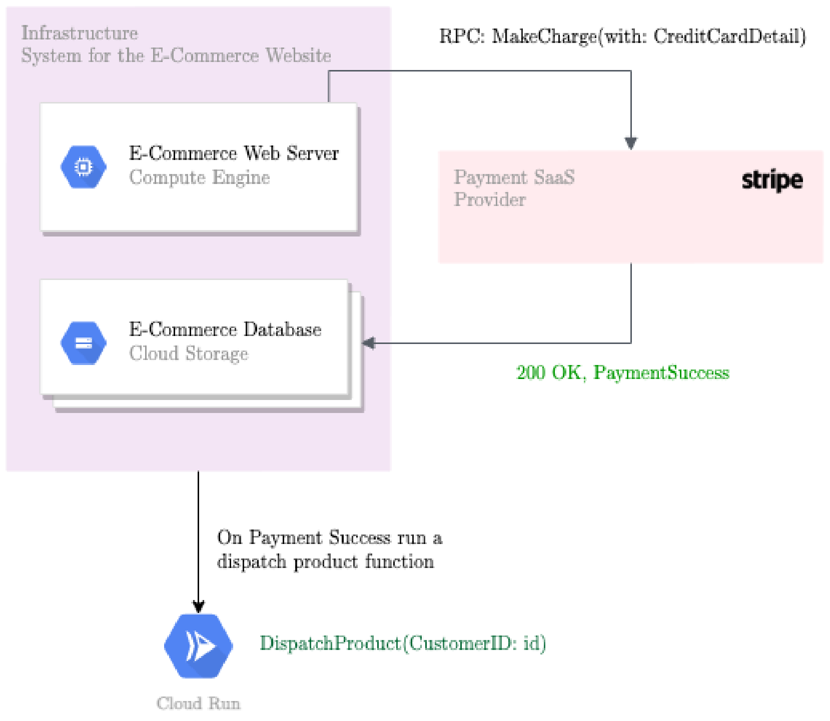
- You can use an industrial scale bandwidth TCP network and a FTP server to deliver these contents with maximum of 4 Gbps speed.
- But you can load all the data in hard disks get it to a plane and deliver the data in a day with all the major courier services. That's like 4728 Gbps speed delivery. This is more reliable than TCP, virtually no loss of data bytes.

Chapter 2

Concepts to build Distributed Systems

2.1 Remote Procedure Calls

Remote procedure calls are function calls that are not part of the main node, instead, the node calls another node running somewhere else and gets the result back. Consider the following system for an e-commerce website, that uses stripe for payment processing:



This e-commerce website is running on Google Cloud and calls `MakeCharge()` function call to stripe's server. This calling a function that is on a different node is called a remote procedure call.

This `MakeCharge()` function logic and implementation are handled by Stripe, they contact the VISA or MasterCard or RuPay server, those payment processors call the Bank to verify credentials and debits the money then VISA or MasterCard server calls stripe to say the payment is OK then stipe calls our e-commerce website that the payment is processed. The banks have different servers, the VISA, MasterCard companies have different servers, stripe as well as our e-commerce website are running on different servers.

This is the best thing about RPCs. We don't have to directly talk to the bank to debit money for the customers' accounts we will call some RPC and we don't have to handle the rest of the logic. These RPCs can call servers belonging to different companies or even the same companies owned by a different engineering team. This creates an abstraction of code and logic in our programs. In case the stripe server fails, our website remains unaffected for simple product browsing and virtually nothing happens to the customers ordering by cash on delivery.

2.1.1 RPC marshalling

To communicate with each other, the servers may need to make available a common message protocol. Let's say our web server is running **Flask** Python framework and Stripe uses Ruby and CoffeeScript according to TechStack.io. So to communicate with each other stripe and our server must conform to some common message structure. In most cases nowadays we use REST-JSON format to communicate between servers.

2.1.2 Problems with RPCs

In the best-case scenario, all the RPC calls should behave like it's running inside the caller machine, but many problems may arise during RPC calls.

- RPC `MakeCharge()` do not go through
- RPC `MakeCharge()` goes through but the stripe server fails during the execution of `MakeCharge()`
- `MakeCharge()` goes through but the `PaymentSuccess` message is lost or delayed and the e-commerce site to stripe connection times out. This is a bad scenario because the customer money is debited but the cloud run function call `DispatchProduct()` will not run because this function is run after the acknowledgement is store on the database.
- `PaymentSuccess` successfully comes back but the e-commerce server fails during the handling of a successful payment.

2.2 Time, Clocks, Event Orders

In distributed systems clocks are enormously important. Time plays a crucial role in the ordering of events. Some of the other things we need time for are the following

- operating systems at several nodes need time to keep track of CPU usage, utilization, network latency,
- validate and store data with limited time validity [time-based caches]
- record when some event occurred
- determining the order of events in a several node architectures which we'll discuss below

Suppose there are three systems A, B, and C. Now **A** sends a message to B and C that *hey, some systems are down in Europe please redirect to India*, hearing this **B** sends a message to A and C that *No, Indian servers are also down, please redirect all Indian traffic as well as European traffic to North America.*

Now look from the American Server C's perspective. Due to network delay, it may happen that message from **B** arrives first before the message from **A**. This order doesn't make any sense for **C** right? If there is some way to order those messages... Oh, we can put some timestamp to each message, so that we can later decode what is the order of received messages and process those messages in their intended order.

But in distributed systems there is no concept of **global time**. Many times servers are running across different zones, we can sync time with that server using NTP (Network time protocol). For example, Apple has a network time server that can be reachable at `time.apple.com`. Once we find the time difference between the machine host and the time server the host machine gently adjusts time by running a little slower or faster on the machine to match with the time server. This gradually reduces the time difference over the course of a few minutes. If NTP finds the difference much more it may reject to change the time on a host and stop, leaving a human operator to adjust the clock.

For these above reasons timestamps can't be reliable in distributed systems. It may so happen that timestamp on message $\mathbf{A} \rightarrow \mathbf{(B\ and\ C)}$ > timestamp on message $\mathbf{B} \rightarrow \mathbf{(A\ and\ C)}$. Now the time delta becomes **-ve** meaning again the message ordering flipped.

2.2.1 Physical Clocks and Logical Clocks

There are 2 types of clocks in the distributed systems, one being physical which counts [how many] seconds elapsed and other being a logical clock which counts logical events like [number of communication happened, number of time a connection created or destroyed].

Chapter 3

Key-Value Database

3.1 Introduction

A key value database is a database service that provides a efficient storage for a key-value pair. Every unique key must be paired with one and only key. For example the following may be a key-value database instance for a website.

ip_addr	252.236.1.55
client_port_num	6677
username	theroyakash
mail_id	hey@theroyakash.com
hashed_password	e5de9701e9a15b1bf134db2da57ab9c2

3.2 Requirement Analysis

When designing a Key-Value database we must consider a trade-off between consistency and availability. Let's design our database highly available to make it usable as a cache. So let's draw our little requirements chart:

- Highly available,
- Low latency output
- Each of the entry will be small. Let's bound that to 1-2 KB
- Huge amount of records to be stored.

3.3 Monolith Design vs Distributed Design

Developing a single server key-value storage makes it a single-point failure service. In designing systems we must avoid such single point of failures.

Capacity Constraints: To make the data access faster we may store the entire database as a hashtable in RAM. But storing records in a single server [RAM] do not support for large scale data storage. You can quickly fill up the entire RAM with the database data. Then multiple page-fault and thrashing will make your system a high latency product.

3.4 Distributed Design

A distributed key-value pair storage is also called distributed hash table. From CAP theorem it is impossible to design a distributed system that gurrantees more than 2 of these 3 things: CONSISTENCY, AVAILABILITY, and PARTITION TOLARANCE.

Chapter 4

Basic terms related to distributed systems

4.1 CDN

CDN is a content delivery network service, mostly used for

Chapter 5

Design URL Shortener Service