Designing Distributed Systems that Scale

Harry Bairstow

March 2023

Contents

1 Introduction

Distributed Systems are designed to scale; but not every design is created equal. I decided to choose this as the topic for my research project so I can explore the key problems that distributed systems have and some solutions that have been implemented. I currently work as a Distributed Systems Engineer at WalletConnect while studying Computer Science so the knowledge that I find while researching this topic should allow me to build better software both personally and professionally. Throughout the project I have looked at what is done now and what could be done in the future to improve the next-generation of Distributed Systems and Software supporting them.

2 Distributed vs Decentralised

Before I get into the details of distributed systems I thought I would discuss the difference between a system being distributed and decentralised. So that there is no confusion as many of the problems I discuss hear can be solved by decentralised systems. However, decentralisation isn't the perfect solution as that has it's own set of problems that could be a research project in its own right.

2.1 Distributed

A distributed system has several nodes but they are all controlled by a single "controller" or "primary" node. This would still be a centralised system even though there are many nodes that could be spread across multiple locations.

One of the papers I read to discover more about the state of load balancing had this description for a distributed system.

> In distributed systems, various nodes act autonomously and cooperate with each other, which can achieve the purposes of resource sharing, openness, concurrency, scalability, fault-tolerance, and transparency (Jiang 2015)

which quite aptly sums up the requirements for a system to be considered distributed.

2.2 Decentralised

In a decentralised system there are sever nodes - similar to a distributed system - but each node has the same permissions and responsibility. Typically anyone can create a node in a decentralised system too which allows for even more nodes. The nodes are spread out across the world similar to the distributed system with the one core difference being the responsibility and ownership of nodes.

3 Service Availability

Distributed systems are typically used when high service availability is a critical system requirement. Some examples of systems that might have this requirements are:

- Banking
- Law Enforcement
- Email Services

Often high availability can be achieved by designing stable software and running on specialised hardware but this might not be enough. This section researches the potential problems with availability in a distributed system and some of the potential solutions that have been found.

3.1 Load Balancing

In 2002, a paper from the University of Texas addressed the issue with static load balancing and summed up the problem as:

> Most of the previous works on static load balancing considered as their main objective the minimization of overall expected response time. The fairness of allocation, which is an important issue for modern distributed systems, has received relatively little attention. (Grosu et al. 2002)

Their paper goes on to show how you can use the "Nash Bargaining Solution" (Grosu et al. 2002) to "guarantees the optimality and the fairness of allocation" (Grosu et al. 2002) their solution looks at splitting jobs by class where a class is a definition of the job across multiple nodes in a system in a fair way.

Load Balancing doesn't just have to be done at the hardware level, another common load balancing use case is in web applications where you may have several server all running the same application around the globe and you need to distribute traffic across them all.

3.1.1 Static Network Load Balancing

"Static load balancing algorithms follow fixed rules and are independent of the current server state." (Amazon Web Services n.d.) Some example of static load balancing algorithms are:

- **Round Robin** is where requests are sent to different servers in order e.g. if you had 3 servers the first request would go to server 1, then 2, then 3, then 1, etc.
- **Weighted Round Robin** is similar to Round Robin except you can add a weight to servers i.e. if you have a more powerful server you can send more traffic to that instead of a less powerful server
- **IP Hash** is where "the load balancer performs a mathematical computation" "on the client IP address. It converts the client IP address to a number, which is then mapped to individual servers." (Amazon Web Services n.d.) this could be useful if you want clients to get the same server each time the make a request or there are servers in different regions to provide quick speeds.

3.1.2 Dynamic Network Load Balancing

"Dynamic load balancing algorithms take the current availability, workload, and health of each server into account. They can shift traffic from overburdened or poorly performing servers to underutilized servers, keeping the distribution even and efficient." (Cloudflare n.d.)

- **Resource Based** is where "load balancers distribute traffic by analyzing the current server load" (Amazon Web Services n.d.)
- **Least Connection** is where "the load balancer checks which servers have the fewest active connections and sends traffic to those servers. This method assumes that all connections require equal processing power for all servers." (Amazon Web Services n.d.)
- **Least Response Time** is where "time method combines the server response time and the active connections to determine the best server. Load balancers use this algorithm to ensure faster service for all users." (Amazon Web Services n.d.)

3.2 Plan for Failure

Sometimes the best solution for availability is to plan for system failure. Having a framework in place for when code crashes and being able to get the system back running, below are two examples of where planning for failure can be used differently. Erlang - a programming language from Ericsson - which uses the language's virtual machine and software to be resilient as well as Kubernetes a tool from Google to deploy code in a way that the nodes can manage what should be running and recover any failed jobs.

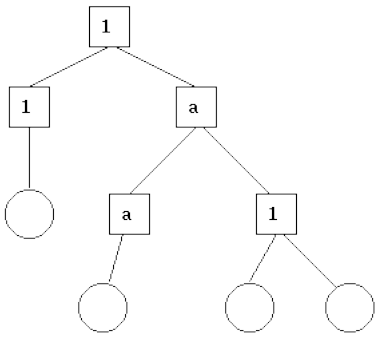
3.2.1 Erlang

"Erlang is a programming language originally developed at the Ericsson Computer Science Laboratory." (Ericsson n.d.*a*) Erlang is made of both the language - with virtual machine - and OTP (Open Telecom Platform) which is a collection of libraries also maintained by Ericsson. "Most projects using "Erlang" are actually using "Erlang/OTP", i.e. the language and the libraries. OTP is also open source." (Ericsson n.d.*b*)

Erlang has a philosophy of "let it crash" (Armstrong 2003) meaning that if the error can be foreseen by the programmer you should you should have protective code written to handle those cases - those are not errors. However, if you have something run that shouldn't have e.g. incorrect function arguments you should just let your Erlang code crash. Erlang uses the Supervisor and worker model which can be summed up as:

> One process, the worker process, does the job. Another process, the supervisor process. observes the worker. If an error occurs in the worker, the supervisor takes actions to correct the error (Armstrong 2003)

Meaning that you end up with a tree of processes similar to the below example from the Erlang documentation.



There is one major negative to Erlang in the modern development space which is its lack of type-safety. There are several solution to solve that but one includes Gleam. Gleam is described as "a functional programming language for writing maintainable and scalable concurrent systems." (Pilfold 2019) - it is a fairly new language but there were two talks this year at FOSDEM (Free and Open Source Software Developers' European Meeting) which were:

- "Introduction to Gleam by building type-safe Discord bots on the BEAM" (Bairstow 2023)
- "Distributed music programming with Gleam, BEAM, and the Web Audio API" (Thompson 2023)

4 Data Consistency

When building distributed systems the issue of data consistency often arises. Where one node has received a request and has not had time to propagate that change of state to all other nodes meaning a request on a different node may then yield different results. There are a few ways to manage this, I have decided to look at Eventual Consistency.

> "Eventual visibility is sufficiently strong to write correct client programs. For example, an eventually visible replicated counter can be used to reliably count events in a distributed systems. However, if eventual visibility is our only guarantee, we may have to deal with a number of confusing anomalies." (Burckhardt et al. 2014)

Eventual consistency means that they entire system will eventually at some point in the future have all nodes state be synced. This is perfectly reasonable if you are willing to accept a few caveats e.g. if Eventual Consistency is your only guarantee to the system you may have anomalies where one node (A) has used an identifier while another node (B) used it too before knowing node A has already used it. You can handle this by using algorithms that prevent such collisions such as Snowflake or CUID.

4.1 Snowflakes

Snowflake's were designed at Twitter to keep up with the amount of tweets being sent in a distributed system as well as having ordering guarantees within K seconds. Where "We're aiming to keep our k below 1 second" (Twitter, inc. 2010). The key requirement was "We needed something that could generate tens of thousands of ids per second in a highly available manner." (Twitter, inc. 2010) and to achieve such they were lead to choose an "uncoordinated approach." (Twitter, inc. 2010). Although there is no-longer an explicit requirement to be within a distributed system this is still a valid solution to data consistency for a distributed system because they have had to manage multiple workers across the system and the only solution that could generate enough identifiers fast enough would be an unconnected one.

5 Conclusion

After looking at the problems that come with Distributed Systems as well as the solutions that are given by various large organisations using them I found that distributed systems can become very complicated very quickly. However, well thought out design allows for the system to scale well and accept everything that is thrown its way. Looking forward I will continue to research this topic and try to pair it with previous research project "Private Set Intersection for Privacy Preserving Comparisons" (Bairstow 2022) to see if there are possibilities to have a truly permissionless distributed systems where all messages between nodes are validated by the controlling node.

References

Amazon Web Services (n.d.), 'What is load balancing?'.
URL: *https://aws.amazon.com/what-is/load-balancing*

Armstrong, J. (2003), Making reliable distributed systems in the presence of software errors, PhD thesis, KTH Royal Institute of Technology.

Bairstow, H. (2022), 'Private set intersection for privacy preserving comparisons'.
URL: *https://harryet.xyz/papers/private-set-intersection.pdf*

Bairstow, H. (2023), Introduction to gleam by building type-safe discord bots on the beam. FOSDEM 2023.
URL: *https://fosdem.org/2023/schedule/event/beam_gleam_intro/*

Burckhardt, S. et al. (2014), 'Principles of eventual consistency', *Foundations and Trends® in Programming Languages* **1**(1-2), 1–150.
URL: *https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/final-printversion-10-5-14.pdf*

Cloudflare (n.d.), 'What is load balancing?'.
URL: *https://www.cloudflare.com/learning/performance/what-is-load-balancing/*

Ericsson (n.d.*a*), 'About'.
URL: *https://www.erlang.org/about*

Ericsson (n.d.*b*), 'What is erlang'.
URL: *https://www.erlang.org/faq/introduction.html*

Grosu, D., Chronopoulos, A. T. & Leung, M.-Y. (2002), Load balancing in distributed systems: An approach using cooperative games, *in* 'Proceedings 16th International Parallel and Distributed Processing Symposium', IEEE, pp. 10–pp.

Jiang, Y. (2015), 'A survey of task allocation and load balancing in distributed systems', *IEEE Transactions on Parallel and Distributed Systems* **27**(2), 585–599.

Pilfold, L. (2019), 'Hello, gleam!'.
URL: *https://gleam.run/news/hello-gleam/*

Thompson, H. (2023), Distributed music programming with gleam, beam, and the web audio api. FOSDEM 2023.
URL: *https://fosdem.org/2023/schedule/event/beam_distributed_music_programming_gleam/*

Twitter, inc. (2010), 'Announcing snowflake'.
URL: *https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake*