Distributed Computing I

March 28, 2022

Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Distributed Systems I

Software Architecture

March 28, 2022

Brae Webb

1 Introduction

We have started looking at distributing our applications. Service-based architectures distribute business processes such that each process is deployed on a separate physical machine [1]. The distributed machines act as if they are the one system, so service-based architectures are our first look at distributed systems.

Definition 1. Distributed System

A system with multiple components located on *different machines* that communicate and coordinate actions in order to *appear as a single coherent system* to the end-user.

Recall that during our investigation of service-based architectures we outlined the following pros and cons:

Simplicity For a distributed system.

Reliability Independent services spreads the risk of fall-over.

Scalability Course-grained services.

Let us take a moment now to reflect deeper on each of these attributes.

2 Simplicity

We said that a service-based architecture was simple 'for a distributed system'. This stipulation is doing a lot of heavy lifting. It can be easy to forget given the wide-spread usage and demand for distributed systems that they are not simple. Let us investigate some logical fallacies that make distributed systems more complicated and less appealing than they might initially appear.

Definition 2. Fallacy

Something that is believed or assumed to be true but is not.

3 Fallacies of Distributed Computing

The first four fallacies of distributed computing were introduced by Bill Joy and Dave Lyon and called 'The Fallacies of Networked Computing' [2]. Peter Deutsh introduced three more fallacies and named the collection 'The Fallacies of Distributed Computing' [3]. James Gosling (lead developer of Java) later introduced a final eighth fallacy [2]. We omit the eighth fallacy as its relevance to the concerns of modern developers has deteriorated.

3.1 The Network is Reliable

The first fallacy assures us that the network is reliable. Developers like to believe that this is a fact which can be relied upon but networks are certainly not reliable. They are more reliable than they used to be but they are far from reliable.

Often we are quite capable of remembering this requirement in obviously unreliable contexts. For example, building a mobile application, it is easy enough to remember that the mobile will occasionally lose a WiFi connection and we should plan for that. Developers can often get the client-server connection correct but neglect server-server connections, assuming they are reliable.

When building distributed systems which need to communicate with each other, we often assume that the connection between servers will always be reliable. However, even in the most tightly controlled network we cannot guarantee that no packets will be dropped. It is important to remember that the reliability of network communication is a fallacy and that we need to build-in error handling between any two machines. We will see later in our distributed systems series that processing network errors can be extremely challenging.[1]

3.2 Latency is Zero

This course is being run in Australia, no one who is taking this course in Australia *should* believe this fallacy. Again, this fallacy doesn't exist as much with client-server communication, we are all intimately familiar with having videos buffer.

This fallacy still lingers with server-server communication. Each time an architect makes a decision to distribute components onto different machines, they make a trade-off, they are making a non-trivial performance sacrifice. If you are designing a system with distributed calls, ensure that you know the performance characteristics of the network and deem the performance trade-offs acceptable.

3.3 Bandwidth is Infinite

Similar to the previous fallacy, the fallacy of infinite bandwidth is a plea for architects to be mindful and conservative in their system designs. We need to be mindful of the consumption of the internal and external consumption of bandwidth for our systems. There are hard limits on bandwidth. A dubious statement from Wikipedia[2] claims that Australia has a total transpacific bandwidth of around 704 GB/s.

Internally, data centers such as Amazon allow impressive bandwidths and it is therefore becoming less of a problem. However, when working at scale, it is important to be mindful to not be wasteful with the size of data communicated internally. Externally bandwidth usage comes at the cost of the end-user and the budget. End-users suffer when bandwidth usage of an application is high, not all users have access to high bandwidth connections, particularly those in developing nations. There is also a cost on the developers end, as infrastructure providers charge for external bandwidth usage.

3.4 The Network is Secure

Developers occasionally assume that VPCs, firewalls, security groups, etc. ensure security. This is not the case. The moment a machine is connected to the internet it is vulnerable to a whole range of attacks. Known and unknown vulnerabilities enable access bypassing network security infrastructure. Injection attacks are still prominent; if a distributed system does not have multiple authentication checkpoints then an injection attack on one insecure machine can compromise the entire system. We tend to incorrectly assume we can trust the machines within our network, and this trust is a fallacy.

[1] https://youtu.be/IP-rGJKSZ3s

[2] https://en.wikipedia.org/wiki/Internet_in_Australia: I cannot find a supporting reference, let me know if you are able.

3.5 The Topology Never Changes

The topology of a network encompasses all of the network communication devices, including server instances. This includes routers, hubs, switches, firewalls, etc. Fortunately, in modern cloud computing we have more control over these network devices. Unfortunately, this control also gives us the ability to rapidly change the topology. The churn of network topologies in the cloud age makes us less likely to make previously common mistakes like relying on static IP addresses. However, we still need to be mindful of network changes. If network topology changes in such a way that latency is increased between machines then we might end up triggering application timeouts and rendering our application useless.

3.6 There is Only One Administrator

This is a fallacy which is encountered infrequently but it is worth pointing out. Imagine you have an application deployed on AWS. To prevent an overly eager graduate developer from taking down your application, your build pipeline does not run on the weekend. Sunday afternoon you start getting bug reports. Users online are unhappy. Your manager is unhappy. You check the logs, there have been no new deployments since Friday. Worse still you can access the application and it works fine. Who do you contact? AWS? The users? Your ISP? Your users ISP's? Who is the mythical sysadmin to solve all your problems? There isn't one; it is important to account for and plan for that. When things start failing can you deploy to a different AWS region? Can you deploy to a different hosting provider? Can you deploy parts of your application on-premise? Likewise we need to be aware of this fallacy when trying to resolve less drastic failures, for example, high latency, permission errors, etc.

3.7 Transport Cost is Zero

When architecting a system it can easy to get carried away with building beautiful distributed, modular, extensible systems. However, we need to be mindful that this costs money. Distributed systems cost far more than monolithic applications. Each RPC or REST request translates to costs. We should also mention that under-utilised machines cost money. Distributing our application services can seem like a beautiful deployment solution but if you are running 10 different service machines for an application with 100 users, you are burning money both hosting the machines and communicating between them.

4 Reliability

We said previously that a service-based distributed architecture was reasonably reliable as it had independent services which spreads the risk of fall-over. We should look a bit more into why we need reliable software, what reliable software is, how we have traditionally achieved reliable software, and how we can use distributed systems to create more reliable software.

4.1 Reliable Software

We want, and in some cases, need, our software to be *reliable*. Our motivation for reliable software ranges from life or death situations all the way to financial motivations. At the extreme end we have radiation therapy machines exposing patients to too much radiation and causing fatalities [4]. On the less extreme end, we have outages from Facebook causing $60 million of lost revenue [5]. Regardless of the motivation, we need reliable software. But what does it mean for software to be reliable?

4.2 Fault Tolerance

In an ideal world we would produce fault-proof software, where we define a fault as something going wrong. Unfortunately, we live in an extremely non-ideal world. Faults are a major part of our software world. Even if we could develop bug-free software, hardware would still fail on us. If we could invent perfectly operating hardware, we would still be subject to cosmic bit flipping[3].

Instead, we learnt long ago that fault tolerance is the best way to develop reliable systems. John von Neumann was one of the first to integrate the notion of fault tolerance to develop reliable hardware systems in the 1950s [6]. Fault tolerant systems are designed to be able to recover in the event of a fault. By anticipating faults, rather than putting our heads in the sand, we can develop much more reliable software.

A part of this philosophy is to write defensive software which anticipates the *likely* software logic faults (bugs). The *likely* modifier is important here, if we write paranoid code which has no trust of other systems, our code will become incredibly complex and ironically more bug prone. Instead, use heuristics and past experience to anticipate systems which are likely to fail and place guards around such systems.

Aside from software logic faults, we have catastrophic software faults and hardware faults. These types of faults cause the software or hardware to become unusable. This occurs in practice far more often than you might expect. Hard disks have a 10 to 50 year mean time to failure [7]. We would only need 1,000 disks to have one die every 10 days. How should we tolerate this type of fault?

4.3 Distributing Risk

For faults which cause a system to become unusable we can not program around it. We need a mechanism for recovering from the fault without relying on the system to work at all as expected. One approach is to duplicate and distribute the original system. If we duplicate our software across two machines then we have halved our risk of the system going completely down.[4] If we replicate our software across thousands of machines then the likelihood of a complete system failure due to a hardware failure is negligible. Google does not go down over a hardware failure. We can imagine that Google servers have many hardware failures per day but this does not cause an outage.

This gives us one very important motivation for creating distributed systems, to ensure that our software system is *reliable*.

5 Scalability

Thus far we have foreshadowed a number of the complexities inherit to distributed systems, we have also reasoned that somewhat counter-intuitively distributing our system can offer us greater overall reliability. Finally, let us shallowly explore the ways cloud platforms can offer us reliable systems via replication.

5.1 Auto-scaling

Auto-scaling is a feature offered by many cloud platforms which allows dynamic provisioning of compute instances. Kubernetes is a non-cloud platform specific tool which also offers auto-scaling. Auto-scaling is specified by a scaling policy which a developer configures to specify when new instances are required, and when existing instances are no longer required.

Auto-scaling policies will specify a desired capacity which is how many instances should currently be running. The actual capacity is how many instances are currently running, this is different from the desired capacity as instances take time to spin up and down. The desired capacity can be based on metrics such as CPU usage, network usage, disk space, etc. configured via the scaling policy. We should also specify

[3] https://www.youtube.com/watch?v=AaZ_RSt0KP8

[4] In a simple ideal world.

minimum and maximum capacity. Our minimum capacity prevents spinning down too many instances when load is low, for example we do not want to have zero actual capacity as a user will need to wait for an instance to spin up to use the service. Likewise the maximum capacity prevents over-spending when load is high.

5.2 Health Checks

In addition to using auto-scaling to respond to dynamic load we can utilise health checks to ensure all running instances are functional. Health checks allow us to build reliable software. A health check is a user specified method for the auto-scaling to check if an instance is healthy. When a health check identifies that an instance is not healthy, it will spin the instance down and replace it with a new healthy instance.

5.3 Load-balancing

An auto-scaling group will allow us to automatically provision new instances based on specified metrics. In combination with health checks we can keep a healthy pool of the correct amount of instances. However, the question arises, how do we actually use these instances together? If we are accessing these instances via network connections we can use a load-balancer. A load-balancer is placed in front of an auto-scaling group. All traffic for a service is sent to the load-balancer and as the name implies, the load-balancer will balance the traffic sent to instances within the auto-scaling group. This allows a group of instances to assume the same end-point with the load-balancer forwarding traffic to individual instances.

6 Conclusion

We have revisited three of the important quality attributes of a service-based architecture. Namely; simplicity, reliability, and scalability. For simplicity we have seen that there are a number of challenges implicit with any distributed system. For reliability we have seen how using a distributed system can increase the reliability of a system. Finally, we have briefly looked at the techniques which make a distributed system more reliable and more scalable. Our treatment of scalability has only scratched the surface when scaling is just straight-forward replication of immutable machines. For the rest of the distributed systems series we will explore more complex versions of scaling.

References

[1] R. Thomas, "Service-based architecture," March 2022. https://csse6400.uqcloud.net/handouts/service-based.pdf.

[2] I. V. D. Hoogen, "Deutsch's fallacies, 10 years after." https://web.archive.org/web/20070811082651/http://java.sys-con.com/read/38665.htm, January 2004.

[3] P. Jausovec, "Fallacies of distributed systems." https://blogs.oracle.com/developers/post/fallacies-of-distributed-systems, November 2020.

[4] N. Leveson and C. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[5] S. Janardhan, "More details about the October 4 outage." https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/, October 2021.

[6] J. van Neumann, "Probabilistic logics and synthesis of reliable organisms from unreliable components, automata studies," vol. 34, pp. 43–98, 1956.

[7] M. Kleppmann, *Designing Data-Intensive Applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, Inc., March 2017.