Distributed Computing III

April 11, 2022

Richard Thomas

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Distributed Computing III

Software Architecture

April 11, 2022

Richard Thomas

1 Introduction

In our introduction to distributed systems we described the fallacies of distributed systems [1]. Some of these fallacies (e.g. the network is reliable, the network is secure and the topology never changes) apply Murphy's Law, *if anything can go wrong it will*, to the context of distributed systems. We will now move on to O'Toole's Commentary, *Murphy was an optimist*.

Large distributed systems consist of thousands of computing platforms, communicating over large distances and over unreliable internet connections. Failure of some part of the system is practically guaranteed [2], the system must be designed to cater for *partial failure*. Even for small systems, some part will eventually fail, so fault handling must be part of the design.

2 Fault Handling

We mentioned that, paradoxically, distributed systems can be more reliable than non-distributed systems because a distributed system spreads risk of failure over multiple machines [1]. This is managed through health checks, load-balancing and auto-scaling. We have also described the use of transactions as a mechanism to deal with some potential failures that affect storage of persistent data [3].

The challenge, particularly when implementing health checks, is determining when a fault has occurred. Most distributed systems communicate over a TCP/IP network. This introduces a layer of uncertainty in trying to determine if a fault exists. A message sent over a TCP/IP network may not be delivered, may be delayed, or the response may not be received. Possible causes of either fault include the following.

- The request sent to another service in the system may not have been delivered.
- The request may be delayed and is waiting in a queue to be processed. (e.g. either the network or the service is overloaded).
- The node running the service may have failed.
- The service may be busy and has temporarily stopped responding.
- The service may have processed the request and replied, but it has not been received.
- The response may be delayed and will be received later.

There are some techniques that can be used to identify some faults, but they are not perfect.

- If a compute node is running and reachable, but does not have a process listening on the destination port, the operating system should close or refuse the TCP connection. This should result in a RST or FIN packet being received by the message sender, with the caveat that the packet may be lost.
- If a process crashes but the compute node is still running, a monitor program running on the node can report the failure to a health monitoring sub-system.

- If a router knows that an IP address is not reachable, it can reply with a destination unreachable packet. But, the router has no additional ways of knowing if an address is not reachable as the rest of the system.
- If the system is running on your own hardware, you may be able to query network switches to detect link failures.

2.1 Retry and Restart

In general, despite the techniques above, the application needs to have a strategy to detect faults and to decide whether to retry a request or that a node is dead. Fault handling has to be responsive in light of the uncertainty of the fault. A general strategy is to retry sending a message a certain number of times and having a time limit. If no response is received within the time limit the system will then decide that the node is dead, will spin up a new node, and remove the dead node from the load balancer's list of active nodes.

The challenge with this strategy is deciding how many retry requests and how long to wait. Multiple retries can swamp an already overloaded node, reducing its performance even more or possibly leading to it crashing. In the first lecture on distributed systems we introduced exponential backoff as a mechanism to reduce the impact of retrying requests [4]. For more information about this strategy, see the retry design pattern [5]. Simple exponential backoff can introduce peaks of load around the exponential delay. Jitter can be added to the delay to spread out these peaks [6].

Determining how long to wait before deciding that a node is dead has its own challenges. If the system decides that a node is dead, then all clients who have sent messages to that node, and have not received a reply, will need to resend their messages to other nodes. Waiting too long reduces the system's responsiveness, as processes wait for a the dead node to reply. It may also reduce the system's overall performance as a backlog of requests need to be processed.

Waiting too short a time may lead to prematurely declaring a node dead. If the node is declared dead but it is just responding slowly because of system load, then resending messages to other nodes increases the load on other nodes. This can lead to a cascading failure, where all nodes are overloaded to the point that they are all declared dead. There is an additional problem of declaring a node dead, which is just slow to respond. It will still be processing requests until it is shutdown, but those requests will be resent to other nodes. This leads to the possibility that some actions will be performed twice.

One option to reduce the variability of message delays is to use UDP rather than TCP at the network level. UDP does not retransmit lost packets, which helps reduces the variability of transmission time. The drawback is that the system will need to manage more messages not being received, as it will not have the automated retransmission of packets provided by TCP. It depends on the type of system, which approach is more beneficial. If the system is transmitting financial data, the greater reliability of TCP probably outweighs the reduced message delay of UDP. Whereas a music streaming service will probably find that having less variability of delay is more beneficial than the reliability of TCP. (Receiving an audio packet, after it needed to be played, is pointless.)

2.2 Timing Faults

We introduced the issue of write conflicts, when discussing multi-leader replication [3]. The issue of determining order of events is applicable to more than just writing to a database. Any situation where event order is important across multiple services (e.g. message queues in an event driven architecture), will have similar issues to overcome.

One intuitive strategy for dealing with some cases of determining event order, is to use a timestamp to record when the event was created. For write conflicts, the idea being that the most recent write is the correct value in the case of a write conflict. There are two problems with this strategy. It is likely that the clocks on the different machines will not be perfectly in sync. It is possible that the machine on which the

last write was performed has a clock that is behind the machine with the previous write. If writes occur in close succession, it is probable that some writes will have timestamps indicating the wrong order of writes.

Trying to synchronise clocks on different computers is difficult. Trying to synchronise using Network Time Protocol (NTP) is not reliable. Network transmission time means that two machines that access one NTP server at the same time are likely to get the time result after different lengths of network delays. The clocks on the two machines are also likely to lose or gain time at different rates after their times have been synchronised. There is a Precision Time Protocol (PTP) that can be used for synchronisation in under a microsecond [7], but it takes significant resources to implement in a system.

Another problem is that computer clocks have finite precision. Two events can occur in close enough succession, even on the same machine, that they will end up having the same timestamp.

There are a few strategies that can be applied to deal with determining the order of events, which do not rely on timestamps. Leslie Lamport, who was referred to in the service-based architecture lecture [8], suggested a strategy of using a logical clock to overcome issues of drift between real clocks on different computers [9]. The key idea is that every message sent to a service includes the logical time at which it was sent. The receiver then adjusts its logical time to be later than when the message was sent. *Designing Data Intensive Applications* describes these problems in great detail and suggests some solution options, with their attendant tradeoffs [10].

3 Consensus

3.1 Behaving Nodes

Leaders & Locks

3.2 Byzantine Faults

Byzantine Generals Problem

Idempotent

4 Consistency

4.1 Eventual Consistency

4.2 Linearizability

4.3 CAP Theorem

5 Conclusion

Designing reliable distributed systems is a complex, but manageable, process. These notes have introduced some of the less intuitive issues that arise in distributed systems and how to design the system to work in the presence of these issues.

It is possible to go a step further and prove the correctness of distributed systems. This involves creating a model of the system and every service in the system. Assumptions about system behaviour can be stated within the model. The algorithms used to implement the system can be proven to work within the system model and its assumptions. Of course, if the assumptions are broken, then any proofs are invalid. CSSE7610 Concurrency: Theory and Practice[1] provides the background knowledge required to perform these proofs and introduces some of the initial modelling techniques required for distributed systems.

[1] https://my.uq.edu.au/programs-courses/course.html?course_code=csse7610

References

[1] B. Webb and R. Thomas, "Distributed systems I," March 2022. https://csse6400.uqcloud.net/handouts/distributed1.pdf.

[2] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan & Claypool, 3rd ed., October 2018.

[3] B. Webb, "Distributed systems II," April 2022. https://csse6400.uqcloud.net/handouts/distributed2.pdf.

[4] B. Webb, "Distributed systems I slides," March 2022. https://csse6400.uqcloud.net/slides/distributed1.pdf.

[5] R. R. Singh, "Understanding retry pattern with exponential back-off and circuit breaker pattern." https://dzone.com/articles/understanding-retry-pattern-with-exponential-back, October 2016.

[6] M. Brooker, "Understanding retry pattern with exponential back-off and circuit breaker pattern." https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/, March 2015.

[7] D. A. (working group chair), *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE Standard Association, 2019 ed., June 2020.

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

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

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