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 these faults 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. Synchronising 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 of under a microsecond [7], but it takes significant resources to implement.

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

In a distributed system, consensus is when a set of nodes in the system agree on some aspect of the system's state. Achieving consensus is a difficult problem, in light of the issues described in section 2. When designing a distributed system, you can take advantage of the abstraction of consensus to ignore the faults it handles. Abstractions like consensus and transactions [3] make it easier to reason about the behaviour of a distributed system. In this section, we will look at how consensus can be implemented.

3.1 Distributed Transactions

A distributed system that needs to enforce a transaction across multiple compute nodes needs a mechanism to gain consensus between all nodes participating in the transaction that it can be committed. This is called the *atomic commit* problem, based on the idea of transaction atomicity from ACID [3].

In a database, a distributed transaction may happen when the database is partitioned and data in different partitions are part of the transaction. It will also happen if the database has a secondary index on a different node to the primary index. Distributed transactions may also occur where events or messages are part of the transaction. An example of this is sending a message to a message queue, while storing data in a database. In the Sahara eCommerce system, when an order is placed it is stored in the order table in the database and a message is placed in the notification queue to notify the customer that the order has been successful. Storing the order may be a transaction in only a single database partition, but to ensure the notification message is only added to the queue if the order is successful requires a distributed transaction.

3.1.1 Two-Phase Commit

A two-phase commit algorithm is a type of consensus algorithm that solves the atomic commit problem. As the name implies, it splits the commit into two steps. There are better consensus algorithms, but two-phase commit is fairly simple and still commonly used.

A *transaction manager* is introduced to coordinate the two-phase commit process. The database nodes that are part of the transaction are called *participants*.

The process starts when some part of the system (`Client` in figure 1) requests to start a distributed transaction and receives a unique transaction identifier (`transId`). The client starts *single-node* transactions

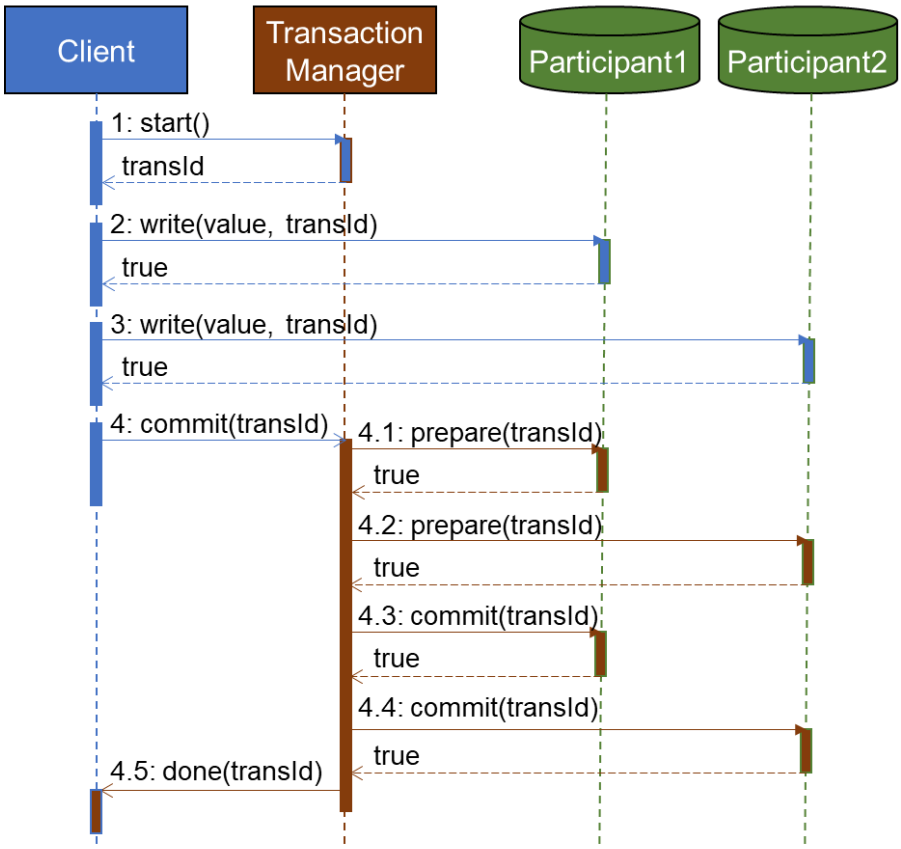

Figure 1: Two-phase commit.

on each, attaching the `transId` to the transaction. The client can perform multiple reads and writes as part of the transaction.

When the client is ready to commit the transaction it notifies the `TransactionManager` to perform the commit. The `TransactionManager` sends a prepare message to all the participants. If a prepare message times out, or any participant reports a failure, the `TransactionManager` send an abort message to all participants and reports the failure to the client.

When a participant receives the prepare message, it confirms that it can commit the transaction, regardless of any fault that may occur after replying to the prepare message. (i.e. The participant has saved all changes as part of this transaction in persistent memory.) The participant is no longer able to abort the transaction itself, it can only abort it now if it receives an abort message from the `TransactionManager`. In effect, it is a pseudo-commit.

When the `TransactionManager` receives positive responses from all participants, it saves the commit decision to a persistent transaction log. This means that even if the `TransactionManager` fails, it can recover the decision. This is called the *commit point*. If the `TransactionManager` decides to abort the transaction, it records the abort decision in the transaction log.

After the commit point has been reached, the `TransactionManager` sends the commit message to all participants. This is an irreversible decision. If the commit message fails or times out, the `TransactionManager` must continue retrying the message until all participants report that they have commited the transaction. This means that even if a participant crashes before it performs the commit, it will complete the commit once the participant restarts. That was the reason for performing a pseudo-commit at the prepare stage, if necessary, the participant can recover and redo all steps that are part of the transaction leading up to the commit.

If the `TransactionManager` fails during the commit process, it can recover and resend the commit messages to any participant that had not responded before the `TransactionManager` failed. If a participant had commited the transaction before the `TransactionManager` failed, it can ignore the new commit

and just report that the commit was successful.

A two-phase commit is called a *blocking* atomic commit, as completion of the commit can take a long time if a participant or the `TransactionManager` fail. Even worse, if the transaction log on the `TransactionManage` or a participant are corrupted when they fail, this may lead to rows in the database never being unlocked. This usually requires manual intervention to decide whether to complete the commit or roll it back, to maintain atomicity. Some libraries that support two-phase commit can make an automated decision in this case, but the decision will probably break atomicity.

Two-phase commit provides a simple abstraction that allows distributed systems to perform safe transactions. The drawback is that a blocking atomic commit is a significant performance cost. Even if the `TransactionManager`, and no participant, fails; rows across multiple database nodes are locked until the commit completes. If there is a failure, recovery can take a long time. For example, in MySQL it is reported that distributed transactions are ten times slower than single-node transactions [11]. Consequently, most NoSQL databases and many cloud-based relational databases do not support distributed transactions.

3.2 Behaving Nodes

Leaders & Locks

3.3 Byzantine Faults

Byzantine Generals Problem

Idempotent

4 Consistency

Consistency is the simple idea that different parts of a system agree on the same value for a data element. Implementing consistency is much more complex than the idea, due to many of the faults that are described in section 2.

4.1 Eventual Consistency

Eventual consistency is a *weak* guarantee. What it guarantees is that all replicated versions of the database will eventually have consistent values for all data they store. The issue is the time it takes to synchronise data. It provides no guarantee of how long until reading a specific value from all replicated databases will return the same value. A read that occurs after a write, but which reads from a replica that has not been updated, will retrieve the old (*stale*) value.

Figure 2 demonstrates the synchronisation issue with an example of searching for a "Nick Cage reversable pillow" from the Sahara on-line store. The product database is replicated to two followers. Searches started after the product has been added to the store may still return that it is not available, until all replicas have been updated.

This leads to the potential for subtle errors in the system logic. It is difficult to think about the consequences of retrieving stale data, as most programmers' intuition is based on the experience of modifying a value and always retrieving the new value. The system design needs to take into account that *all* database reads are performed under the assumption that the value may be stale. Testing for errors caused by data not being consistent is difficult, as you have to force data inconsistencies.

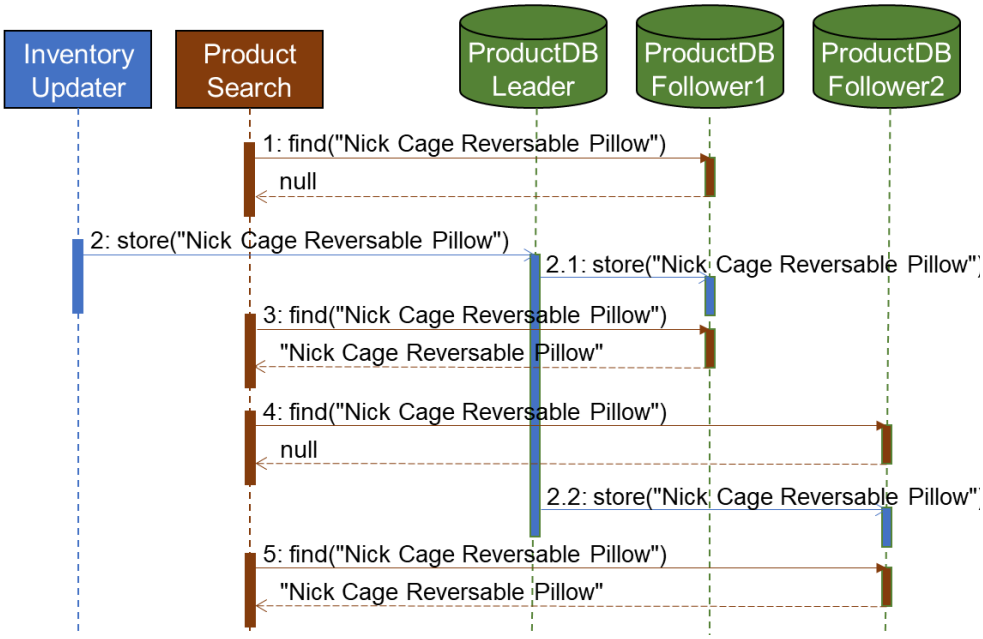

Figure 2: Eventual consistency issues.

4.2 Linearisability

Contrary to eventual consistency, linearisability is a *strong* guarantee. A database system that implements linearisability provides an abstraction layer that allows clients who use the database to work as if there is only a single instance of the database, regardless of how many replicas have been deployed. This provides a simple interaction model that corresponds to expectations from non-distributed systems.

Linearisability means that once a client has written data to the database, all clients who read the that data will see the same value, regardless of the replica from which they read the data. The challenge is implementing linearisability.

4.2.1 Application

Situations where linearisability can be useful include when uniqueness needs to be guaranteed. For example, when a customer registers as a member of the Sahara eCommerce system, they need a unique identifier. If two people attempt to register at the same time and select the same user id, the system needs to be able to linearise the requests to return an error message to one of the users. The same issue arises in banking applications. If a user transfers money from their account at the same time as an automated payment occurs, the account debit service may need to guarantee that the account balance does not become negative. It needs to be able to linearise these two operations and disallow one if it would result in a negative balance.

In some situations, it is acceptable to not require linearisability of operations. If two customers order the last item of a product in the Sahara eCommerce system, the stakeholders and designers may decide that they will allow both orders to proceed. The resolution may be that when the fulfillment service or the warehouse discover that the product is out of stock, they generate an event that changes one of the customers' orders to become a backorder.

The simplistic implementation of linearisability is to have a single database. This defeats the purpose of replicating a database to improve performance and reliability.

4.2.2 Single-Leader Replication

Single-leader replication can be implemented in such a way as to provide linearisability. This is implemented by only allowing reads to be from the leader or from followers that are synchronously updated by the leader. This means that the replicated databases can guarantee that all reads after a write will return the current value.

Forcing all reads to be from the leader defeats one of the purposes of replicating a database, which is the performance benefit of performing queries on followers and only using the leader for update operations. Synchronous updates of at least some followers defeats the performance benefit of asynchronous communication, but at least allows queries on followers. Either of these approaches maintain the reliability advantage of replicating the database.

An additional complication to implementing linearisable single-leader replication is determining which replica is the leader. A typical approach is to have a lock of some form that is acquired by the leader, and all other replicas are followers. Acquiring the lock tself must be a linearisable operation. If more than one replica believes it is the leader, it means that write behaviour cannot be linearisable. A coordination service, such as etcd[1], can be used to implement distributed locks in a linearisable fashion.

4.2.3 Multi-Leader Replication

Multi-leader replication cannot provide linearisability, without introducing so many constraints as to eliminate any benefit of using the approach in the first place. This is because multi-leader replication allows concurrent writes to multiple leaders, and the data is asynchronously replicated to the followers and other leaders.

4.2.4 Leaderless Replication

It is difficult to provide linearisability with leaderless replication. Quorum reads [3] do not guarantee linearisability. With asynchronous communication and network delays it is possible for a write to start, but for a concurrent read to obtain stale data from all members of the quorum that were queried. This occurs when a read retrieves a value from a database that has not yet been written to, but after the write operation started and other databases were updated. The read obtains a consistently stale value, despite obtaining the data from a quorum of databases.

It is possible to provide linearisability using strict quorums with leaderless replication. This comes at the cost of performance, just as it does with single-leader replication. Writes must read the state of a quorum of databases and obtain a lock on the value before performing the write. This allows the write to be performed synchronously.

In leaderless replication, only basic read and write operations are linearisable. More sophisticated atomic operations or transactions require a consensus algorithm to be used to provide linearisability.

4.2.5 Consensus Algorithms

Consensus algorithms, such as XXXXXX, have some similarity to single-leader replication. Their internal implementations prevent stale replica data or having multiple leader nodes. The previously mentioned coordination service, etcd, implements its own consensus algorithm, that provides a key-value store that guarantees linearisability.

4.2.6 Consequences

Linearisability means that if some database replicas lose network connection to the rest of the system, they cannot process any requests. Writes cannot be made as they cannot be linearised. Reads connot be

[1] https://etcd.io/

made as they may obtain stale data.

Consequently, systems, or parts of systems, that require linearisability demonstrate lower availability due to potential network faults. This observation is known as the CAP theorem [12]. Distributed systems can have *consistency* in the presence of network *partitioning*. Or, they can have *availability* in the presence of network *partitioning*. The ideas have been around since the mid 1970's [13], despite the much more recent reference to the CAP theorem. The prime value of the CAP theorem is how it influenced the development of NoSQL databases.

4.3 Causal Ordering

Linearisability is one approach to defining the order in which read and write operations are conceptually performed. It defines a total order[2] on the operations. This means that operations can be compared to each other to determine which should be considered to have occurred first. Or, from the perspective of the linearisability abstraction, there are no concurrent operations on the database.

Another approach to defining the order of operations is to define a *partial* order based on causality. That is the ordering is based on one operation occurring before another (i.e. they are causally related), but other operations may be concurrent. A Git repository's history demonstrates causal dependencies. Within a single branch commits occur in sequential order. But, with multiple branches, development on each branch progresses in parallel with the others and commits may occur concurrently on different branches.

Causal ordering is not as strict as linearisability, which in turn results in less performance cost to implement causal ordering. The issue is to capture causal dependencies. This requires a mechanism to determine which operation happened before another one. The consequence is that the operations must be performed in the same order on all replicas. If causal ordering is not required for some operations, they may occur concurrently and it does not matter if they execute in a different order on a replica.

A single-leader replicated database can record an increasing sequence number with every write operation it records in its log. Followers can then read the log to execute writes in sequence number order. All followers will then be causally consistent with the leader. The drawback is that a single-leader cannot be scaled if asynchronous writes occur faster than it can process them. There is also the difficulty in handling the situation where the leader fails and selecting a new leader.

4.3.1 Lamport Timestamps

If the system does not have a single-leader, the logical clock approach mentioned in section 2.2 provides a mechanism to define causal ordering. This approach is called *Lamport timestamps* [9]. Each node has an id and counts the number of operations it has executed. The timestamp is a tuple (*counterValue*, *nodeID*). The *nodeID* guarantees that every timestamp is unique, even if they have the same counter value. Every node stores the *maximum* counter value it has seen so far. This maximum is passed in every request to another node. If a node receives a request or response with a maximum counter value greater than its own counter value, it increases its own counter to the new maximum.

In figure 3, when `InventoryUpdater` sends message `4` to `ProductDB2`, the message includes a parameter which is `InventoryUpdater`'s current maximum value, which is `1`. `ProductDB2`'s counter value is already `2`, so it increases it to `3` and returns it as part of the Lamport timestamp to `InventoryUpdater`. `InventoryUpdater` records `3` as its new maximum value. When `InventoryUpdater` sends message `5` to `ProductDB1`, the message includes `InventoryUpdater`'s maximum value of `3`. `ProductDB1`'s counter value was `1`, so it increases it to `4` and returns it to `InventoryUpdater`.

Lamport timestamps define causal ordering for operations but they do not guarantee consistency when an operation is performed. If two orders are sent to `ProductDB1` and `ProductDB2` at the same time and there is only one item in stock, Lamport timestamps provide a way to decide which order occurred first, but only after the orders are processed.

[2] https://mathworld.wolfram.com/TotallyOrderedSet.html

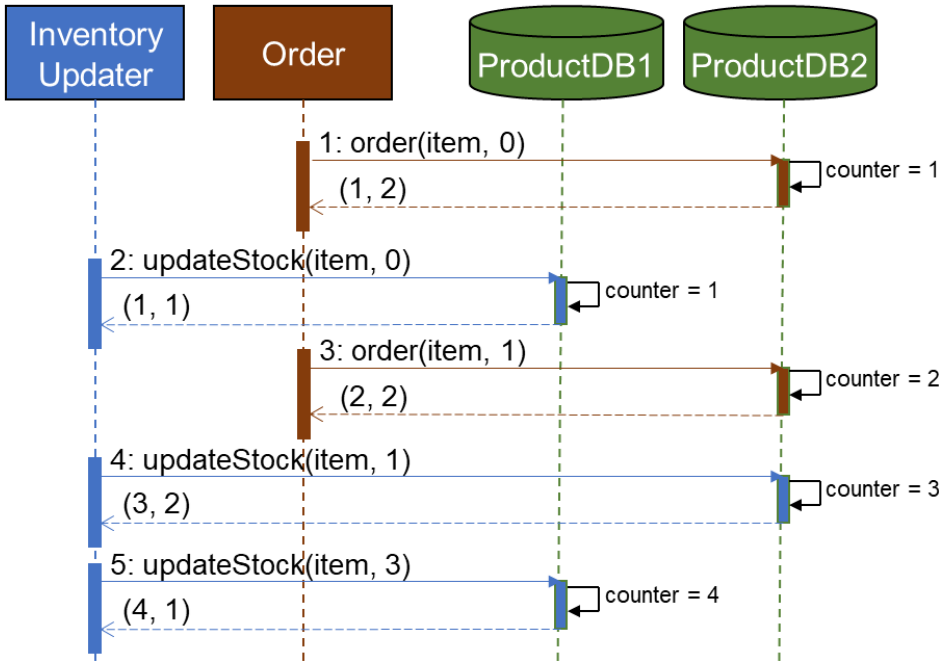

Figure 3: Lamport timestamps.

To enforce a constraint that would not allow two orders for the last item in stock, there needs to be a mechanism to indicate when the order is completed. *Total order broadcast* is one technique that does this. Messages are ordered by their delivery time, not when a result is returned. So, if a message has been received, another message cannot be performed before it. Coordination services, such as etcd[3], implement total order broadcast.

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[4] provides the background knowledge required to perform these proofs and introduces some of the initial modelling techniques required for distributed systems.

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] https://etcd.io/

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

[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.

[11] R. Wigginton, R. Lowe, M. Albe, and F. Ipar, "Distributed transactions: A primer with mysql," in *MySQL Conference and Expo*, (Santa Clara, CA), April 2013.

[12] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[13] P. R. Johnson and R. H. Thomas, "Rfc 677: The maintenance of duplicate databases." https://www.ietf.org/rfc/rfc677.html, January 1975.