Distributed Systems II

April 4, 2022

Brae Webb

Presented for the Software Architecture course
at the University of Queensland

THE UNIVERSITY
OF QUEENSLAND
AUSTRALIA

Distributed Systems II   Software Architecture

April 4, 2022   Brae Webb

1   Introduction

In the previous course notes [1] and lecture [2] on distributed systems, we explored how to leverage distributed systems to increase the reliability and scalability of a system. Specifically, we saw that when working with stateless services which do not require persistent data, auto-scaling groups and load-balancers can be used to scale-out the service — distributing the load to arbitrary copies of the service. In the lecture, we applied this technique to the product browsing service of our Sahara example, as illustrated in Figure 1.

Figure 1: A service-based implementation of Sahara with stateless scaling techniques applied to the product browsing service.

One might correctly identify that by scaling the product browsing service, we have only increased the maximum load that the database can expect. The database becomes a major bottle-neck. We might

attempt to scale-up the database, purchasing more powerful machines to host the database on, but this approach is limited.

In this part of our distributed systems series, we look at how to scale stateful services which *do* require persistent data. We focus on state stored in a database and thus, how databases can be made more reliable and scalable. In these lecture notes we only outline the scope of our treatment of database scaling. For a detailed treatment of these topics, please refer to the Designing Data-Intensive Applications textbook [3].

2 Replication

Replication is the most straight-forward approach to scaling databases. In most applications, read operations occur much less frequently than write operations, in such cases replication can enable improved capacity for read operations whilst keeping the write capacity roughly equivalent. Replication involves creating complete database copies, called replicas, that reduce the load on any single replica.

2.1 Leader and Followers

The most common implementation of replication is leader-follower replication, fortunately, it is also the simplest.

Leader a replica that accepts write operations and define the order in which writes are applied.

Follower replicas are read-only copies of the data.

When writes occur, they must be sent to the leader replica. Leader replicas propagate updates to all followers. Read operations may occur on any of the follower replicas.

For our example, we might create two followers, or read replicas, of the Sahara database. As the product browsing service is likely to primarily perform read queries, it may send those requests to one of the read replicas. Write operations, which are likely to be the majority for the product purchasing service, must still be sent to the lead replica[1]. The resulting system might look like Figure 2.
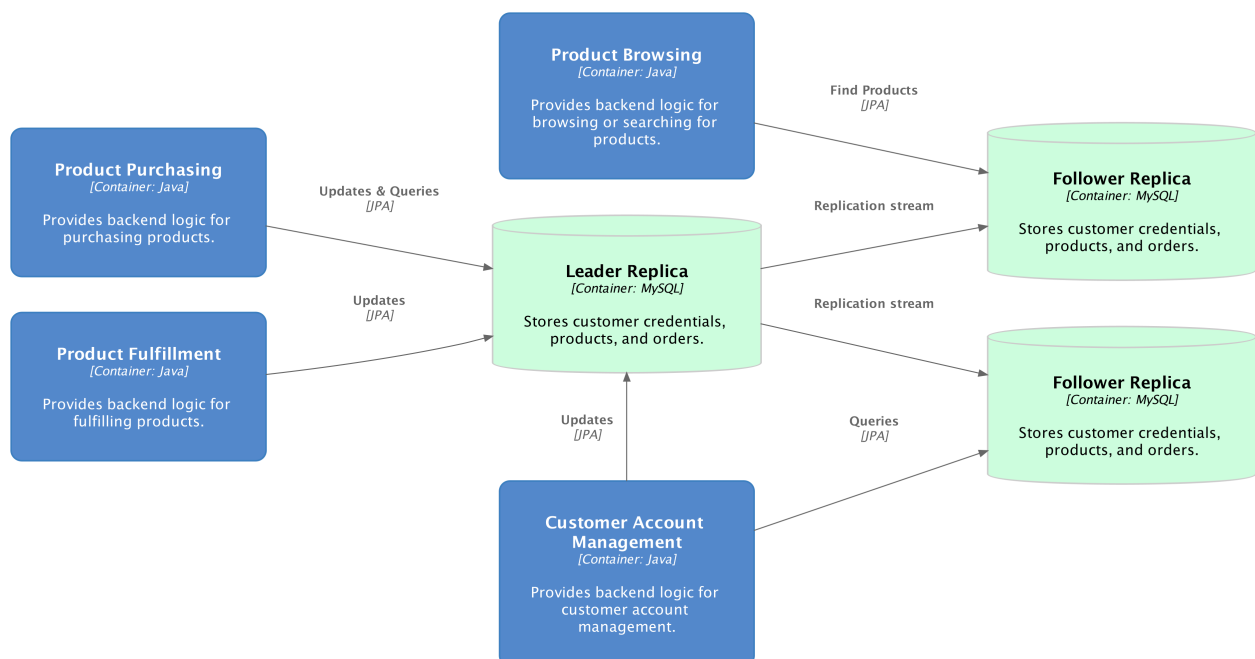

Figure 2: A leader-follower replication of the Sahara database.

---

[1]Realistically, queries would be forwarded based on their type rather than their service of origin.

Replication Lag

For leader-follower replication to be practical, write updates from the leader to the followers need to be *asynchronous*[2]. However, asynchronous propagation introduces an entirely new suite of issues. If we do not wait for write updates to be propagated to all replicas, then there will be times where the data retrieved from a read replica will be out-dated, or *stale*. Eventually, all writes will be propagated to every read replica, so we call this an *eventually consistent* system.

We have grown fairly accustomed to eventually consistent systems. Consider the last time you or a friend of yours updated their display picture. Did everyone start seeing the updated picture immediately, or, did it take a number of hours or even days to propagate through? Despite our increased exposure to eventually consistent systems, there are a few common practices to keep in mind when implementing such a system to preserve your users sanity.

Read-your-writes Consistency Ensures that the user who wrote the database change will always see their update reflected, though other users will still have to wait for the change to be propagated. This type of consistency avoids upsetting users and making them think they must redo their write.

Monotonic Reads Ensures that once a user reads the updated data, they do not later see the older version of the data, i.e. they don't travel back in time.

Consistent Prefix Reads Ensures that sequential writes are always read in the same order. Consider a Twitter feed, each post is a write. Consistent Prefix Reads guarantees that those readers do not see the posts out of order.

2.2   Multi-leader Replication

Leader-follower replications are sufficient for most use cases as reads often occur far more frequently than writes. However, there are situations where leader-follower replication is insufficient. For systems which need to be highly available, having a single point of failure, the leader replica, is not good enough. Likewise, for systems which need to be highly scalable, a write bottle-neck is also insufficient. For these situations, a multi-leader replication scheme may be appropriate. It is worth noting that multi-leader replications introduce complexity that often outweighs the value.

For our example, we might naively introduce a system such as Figure 3. Here we have introduced a second leader, and each leader has their own follower. This type of separation might make sense. We might find it beneficial to have a separate database replica in the warehouse which can be interacted with via the fulfillment service. Such a system would isolate the warehouse operations team from the potential latency of the external customer load.

However, we now have a system where writes can occur in parallel. This can cause several problems. Consider a situation where the fulfillment center has noticed a defective Nicholas Cage Reversible Pillow. They promptly update their system to reflect the decreased stock. However, at around the same time, the CSSE6400 tutors placed a bulk order for all of Sahara's remaining stock. Both write operations appear successful to the fulfillment team and the tutors but a conflict has occurred — this is known as a *write conflict*, and it is a common problem in multi-leader replications.

Write Conflicts

Write conflicts occur when two or more leaders in a multi-leader replication have updates to the same piece of data, in relational systems this is normally the same table row. There are a few mechanisms for resolving write conflicts, but the recommended approach is to avoid conflicts altogether. Conflicts can be

---

[2]We need not wait for writes to propagate to all followers before accepting that a write has succeeded

Figure 3: A multi-Leader replication of the Sahara database.

avoided by ensuring that a piece of data is only ever updated by the same leader, for example, we might implement that all Nicholas Cage related products are written to Leader Replica 1.

If we are not fortunate enough to be in a situation where conflicts can be avoided, there are a few techniques we can use.

- Assign IDs to writes, and decide which conflicting write to accept based on the ID via some strategy (i.e. highest ID). This results in lost data.

- Assign an explicit priority to leader replicas. For example, if there is a merge we might accept Leader Replica 1 as the source of truth. This also results in lost data.

- Merge the values together via some strategy. This works for data such as text documents but is inappropriate for stocks of products.

- Application code resolution. As most conflict resolution is application dependent, many databases allow users to write code which can be executed on write of a conflict or read of a conflict to resolve the conflict, where appropriate our application could even ask the user to resolve the conflict.

2.3   Leaderless Replication

Leaderless replication doesn't rely on writes being sent to and processed by a single designated leader, instead reads and writes may be sent to any of the database replicas. To accomplish this, leaderless databases introduce a few additional constraints on both read and writes operations to maintain relative consistency.

A core aspect of leaderless replication is that clients need to duplicate both write and read operations to multiple replicas. This may be implemented by each client communicating directly with the database replicas, as in Figure 4, or one of the replica nodes can act as a coordinator node and forward requests to other database replicas, as in Figure 5.

4

Figure 4: A leaderless replication of the Sahara database.

Figure 5: A leaderless replication using a coordinator node of the Sahara database.

In leaderless databases, any database node may go offline at any point and the system should continue to function, assuming only a few nodes go offline. This type of database gives us excellent reliability and scalability as we can keep adding extra replicas to increase our capacity. But how can we support this?

Quorum

To support the extensibility of leaderless databases, we need to duplicate our read and write requests. As a simple example, take Figure 4, we have three replicas of our database. When making a write request, if we send our write to just one replica then at most two replicas can have outdated data. Therefore when we read, we need to read from all three replicas to ensure one will have the most recent writes. If instead, we write to two replicas then at most one replica will have outdated data. Then we need only read from two replicas to ensure that at least one will have the most recent writes.

To generalize, if we have a leaderless database of n replicas, we need to satisfy that,

$$w + r > n$$

where w is the amount of replicas to write to and r is the amount of replicas to read from. By satisfying this equation we have quorum consistency, the set of writes and reads must overlap, which means that reading stale data is *unlikely*[3].

---

[3]The cases where stale data can still be read are enumerated in M. Kleppmann [3].

3 Partitioning

Partitioning involves distributing the contents of a database across multiple nodes. It differs from replication as nodes do not store complete copies of the database. Often partitioning is combined with replication to prevent the lost of data if one of the partitions fails. Partitioning is required when we have a large amount of data to store.

When we start partitioning data onto multiple nodes, we need to consider how we decide which data belongs to which node. There are two primary approaches to this decision.

Partition by Key Range

We assume that data is given a key, if a key-value database this is required, in relational databases the schema should specify a primary key column. One approach to partitioning is to divide the range of key values and partition based on that. Consider the UQ student database, the key would be the student ID. We can then partition the database based on student ID, say all student numbers between `s0000000` and `s1000000` are stored on partition 0, all numbers between `s1000001` and `s2000000` are stored on partition 1, etc.

This approach appears practical, each partition has an equal share of database entires. However, as time goes on certain partitions become less used while others become very popular. We might imagine that at the moment, partitions 5 and 6 would be used far more frequently than partition 0. This is known as *skewed partitioning*.

Partition by Hash

Consider hash maps, we use hashes of the keys to allocate values into buckets. A good hashing algorithm minimizes the amount of collision (a value stored in the same bucket). We can apply the same process to database partitioning, hashing the key of a record allows us to maximize the spread of partition usage. The disadvantage of partitioning by hash is that only queries by key are efficient, queries by ranges of keys are no longer efficient requiring all partitions to be queried.[4]

Secondary Indexes

Databases often rely on secondary index to perform a range of useful queries. In our UQ student database example, a database field which is often useful to query might be the degree of a student record. We might want this query to run efficiently as schools could need to frequently query for these collections to implement their own services.

The two approaches to this are local indexes and global indexes. In local indexes, each partition stores an index for their own partition of the data. In global indexes, a partition is assigned to manage a particular secondary index value. For example, partition 2 might store the index for all Bachelor of Software Engineering students.

Local indexes allow efficient writing, as a write only has to update the index of the partition which it is writing to. However, it introduces slower reads as all partitions must be queried for their index. Global indexes slow down writing as all relevant partitions need to have their partition updated. Although it can increase read speeds as a query only needs to look at the relevant partitions.

3.1 Rebalancing

Over time some partitions may get more frequent use than others. We might also add or remove nodes. This can require the data to be rebalanced to update the range of values stored on any partition. Rebal-

---

[4]Databases such as Cassandra use compound primary keys to help resolve this issue.

ancing can be an automated or a manual process. It is important to consider the effects of rebalancing automatically as they are costly operations and if incorrectly triggered automatically can greatly impact the performance of a database.

3.2   Routing

Much like auto-scaled stateless services, we need a way to route traffic that is fair. However, unlike stateless services, only one node of the system is capable of processing our request. We need a mechanism for choosing the appropriate node to send our request to.

We might implement a naive load balancer which just balances the load regardless of the request. This load balancer is will send requests to the wrong nodes. In such a system, it is the responsibility of the node to forward the request to the node which is capable of answering the query.

A more sophisticated solution would be to deploy a load balancer which is request aware. This load balancer can forward requests to the appropriate node.

Finally, we might offload the problem to the client. Requiring that our database clients are aware of how to map their query to the appropriate node.

4   Transactions

All of the various failure cases and considerations when working with scalable databases can be overwhelming. The complexities of distributed databases make reasoning about the logic of our application challenging, which in turn leads to more bugs and faults. Fortunately, database developers have a technique to assist in this complexity and allow us to reason about our interactions with the database, *transactions*.

Transactions bundle related groups of read/write operations together and guarantee useful properties. Traditionally, these properties are referred to as ACID properties, Atomicity, Consistency, Isolation, and Durability.

4.1   Atomicity

Atomicity guarantees that when performing multiple operations, if one operation cannot be completed successfully then the effects of all operations are aborted. For example, consider process purchasing. When purchasing a product we may wish to both update the stock of the product and reduce the funds of a customers account. If either of those operations occurred on their own, someone would be very unhappy. Atomicity guarantees that either all operations in a transaction succeed or none do.

4.2   Consistency

Consistency guarantees that any application specific invariants of the database are preserved by transactions. For example, we might introduce the invariant that the stock of an item cannot be below zero. This invariant must then be maintained by all transactions.

4.3   Isolation

Isolation allows transactions to pretend that they are the only operation performing on the data. Transactions appear to preform sequentially even though they may in reality be performing concurrently. This allows transactions to ignore a whole host of issues related to concurrency.

4.4 Durability

Durability provides a guarantee that once transactions are completed, their updates are preserved. In distributed databases this is often an assurance that the updated data has been copied to other machines.

5 Conclusion

Scaling out services that have persistent data requires far greater care than stateless services. We have seen approaches for coping with undue load on individual nodes via replication. We have seen approaches for handling greater volume of data than can be stored on a single machine via partitioning. We have noted that often one might want to combine the approaches of replication and partitioning for the one database system. Both of these techniques introduce a new range of complexity for programmers to deal with. In our final section we briefly introduced how transactions and the guarantees they provide can help developers in reasoning about their database interactions.

This lecture note has been a very brief introduction into the topics of replication, partitioning, and transactions. The content has been structured around the Chapters 5, 6, and 7 of Designing Data-Intensive Applications [3]. You are strongly encouraged to read these chapters of the book for a much more in-depth look at all of the topics covered.

References

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

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

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