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

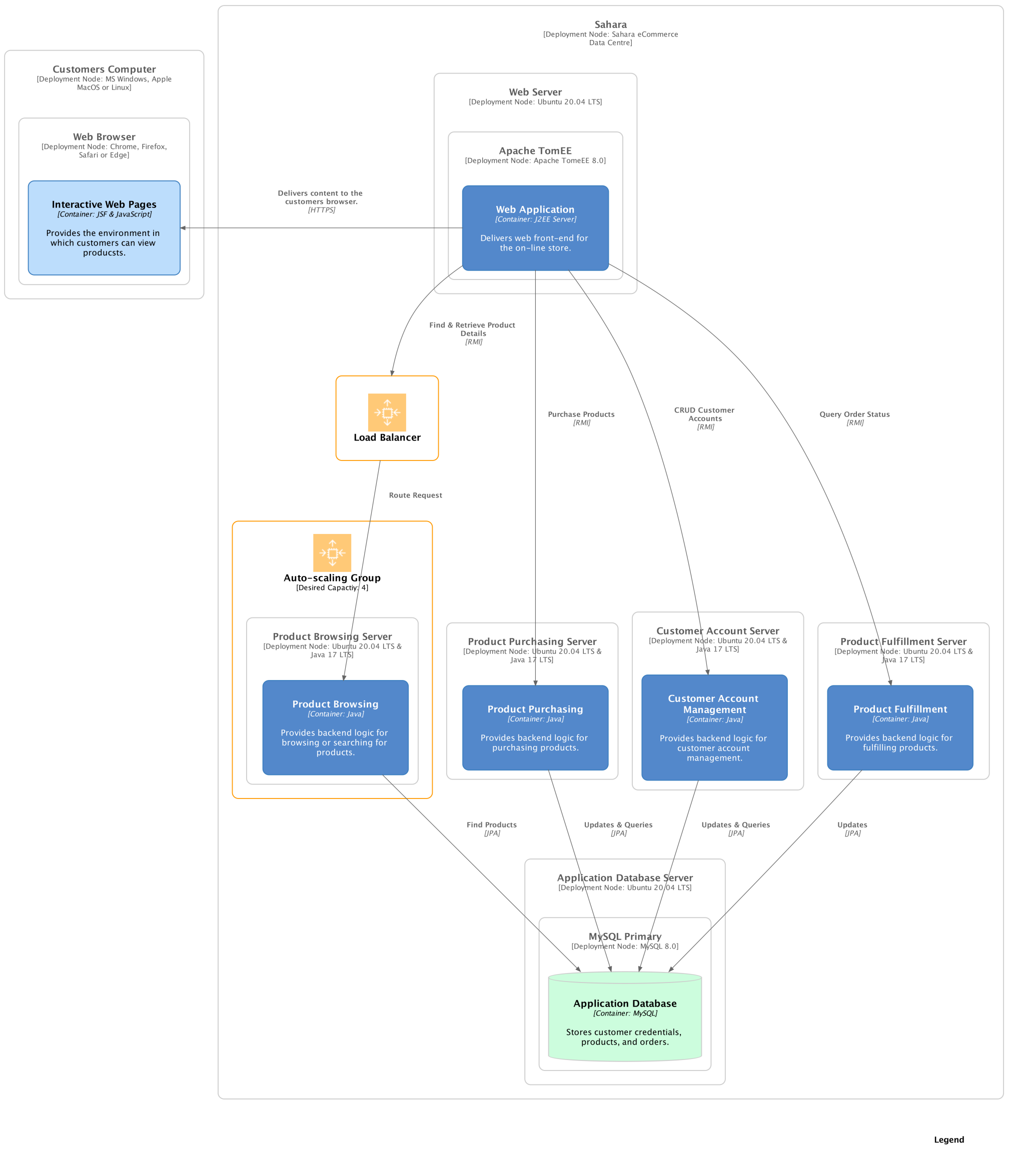
Software Architecture

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.



Sahara
[Deployment Node: Sahara eCommerce Data Centre]
Customers Computer
[Deployment Node: MS Windows, Apple MacOS or Linux]
Web Browser
[Deployment Node: Chrome, Firefox, Safari or Edge]
Interactive Web Pages
[Container: JSF & JavaScript]
Provides the environment in which customers can view producsts.
Web Server
[Deployment Node: Ubuntu 20.04 LTS]
Apache TomEE
[Deployment Node: Apache TomeEE 8.0]
Web Application
[Container: J2EE Server]
Delivers web front-end for the on-line store.
Delivers content to the customers browser.
[HTTPS]
Find & Retrieve Product Details
[RMI]
Load Balancer
Route Request
Purchase Products
[RMI]
CRUD Customer Accounts
[RMI]
Query Order Status
[RMI]
Auto-scaling Group
[Desired Capactiy: 4]
Product Browsing Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Product Browsing
[Container: Java]
Provides backend logic for browsing or searching for products.
Product Purchasing Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Product Purchasing
[Container: Java]
Provides backend logic for purchasing products.
Customer Account Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Customer Account Management
[Container: Java]
Provides backend logic for customer account management.
Product Fulfillment Server
[Deployment Node: Ubuntu 20.04 LTS & Java 17 LTS]
Product Fulfillment
[Container: Java]
Provides backend logic for fulfilling products.
Find Products
[JPA]
Updates & Queries
[JPA]
Updates & Queries
[JPA]
Updates
[JPA]
Application Database Server
[Deployment Node: Ubuntu 20.04 LTS]
MySQL Primary
[Deployment Node: MySQL 8.0]
Application Database
[Container: MySQL]
Stores customer credentials, products, and orders.
Legend

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

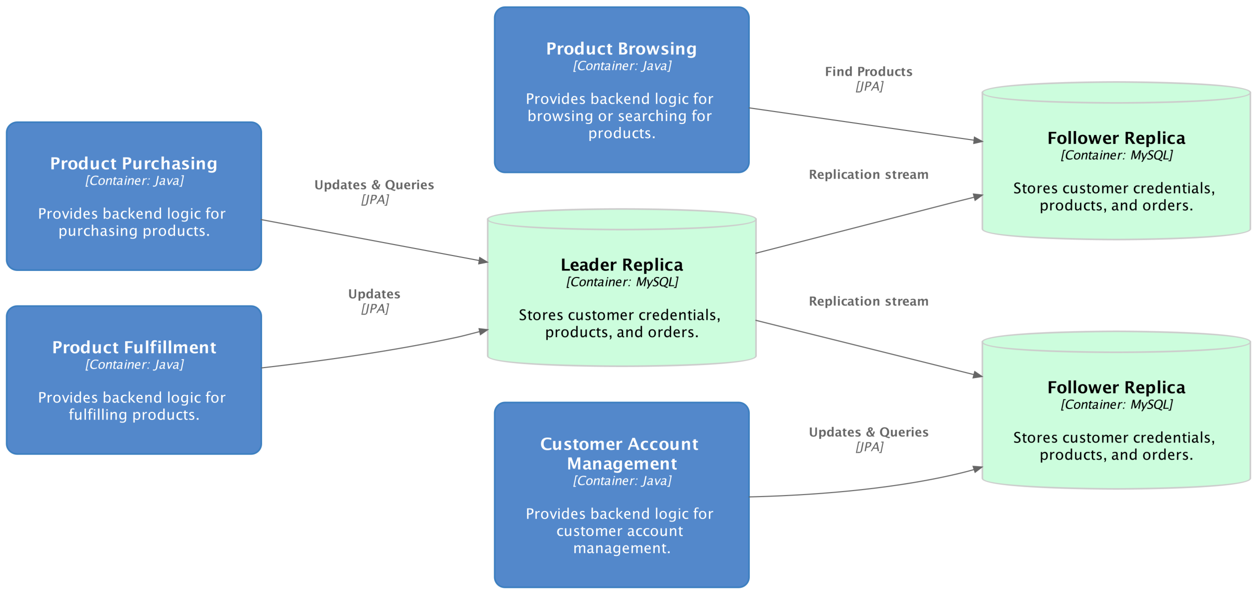
The most common implementation of replication is leader-follower replication, fortunately, it is also the simplest.

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.



Product Browsing
[Container: Java]
Provides backend logic for browsing or searching for products.
Find Products
[JPA]
Follower Replica
[Container: MySQL]
Stores customer credentials, products, and orders.
Product Purchasing
[Container: Java]
Provides backend logic for purchasing products.
Updates & Queries
[JPA]
Replication stream
Leader Replica
[Container: MySQL]
Stores customer credentials, products, and orders.
Updates
[JPA]
Replication stream
Product Fulfillment
[Container: Java]
Provides backend logic for fulfilling products.
Follower Replica
[Container: MySQL]
Stores customer credentials, products, and orders.
Customer Account Management
[Container: Java]
Provides backend logic for customer account management.
Updates & Queries
[JPA]

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 a 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 the around 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

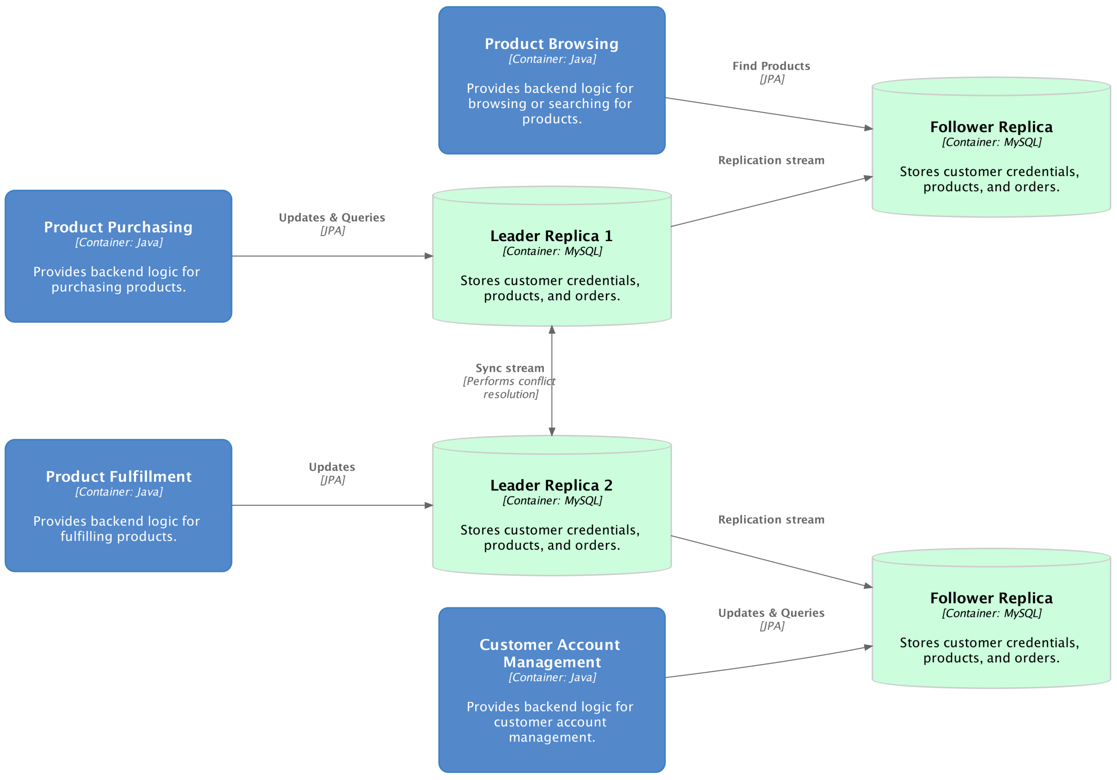

Product Browsing
[Container: Java]
Provides backend logic for browsing or searching for products.
Find Products
[JPA]
Follower Replica
[Container: MySQL]
Stores customer credentials, products, and orders.
Replication stream
Product Purchasing
[Container: Java]
Provides backend logic for purchasing products.
Updates & Queries
[JPA]
Leader Replica 1
[Container: MySQL]
Stores customer credentials, products, and orders.
Sync stream
[Performs conflict resolution]
Product Fulfillment
[Container: Java]
Provides backend logic for fulfilling products.
Updates
[JPA]
Leader Replica 2
[Container: MySQL]
Stores customer credentials, products, and orders.
Replication stream
Follower Replica
[Container: MySQL]
Stores customer credentials, products, and orders.
Updates & Queries
[JPA]
Customer Account Management
[Container: Java]
Provides backend logic for customer account management.

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.

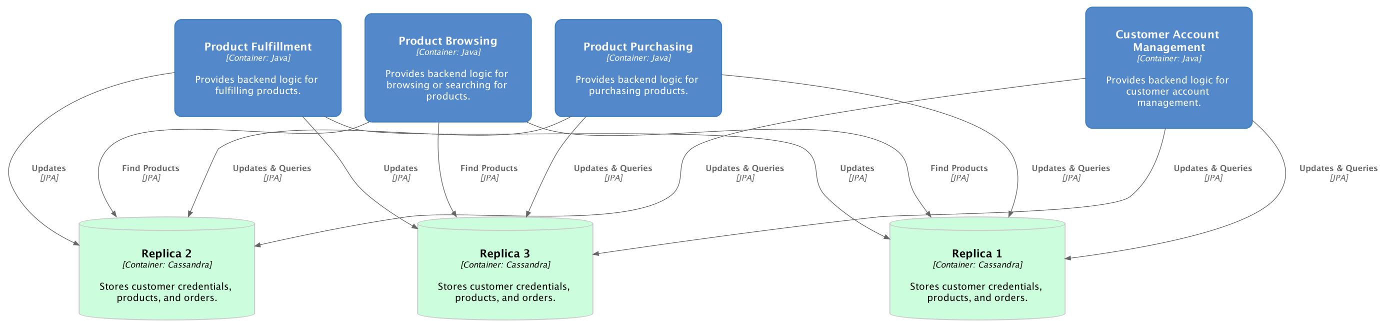


Figure 4: A leaderless replication of the Sahara database.

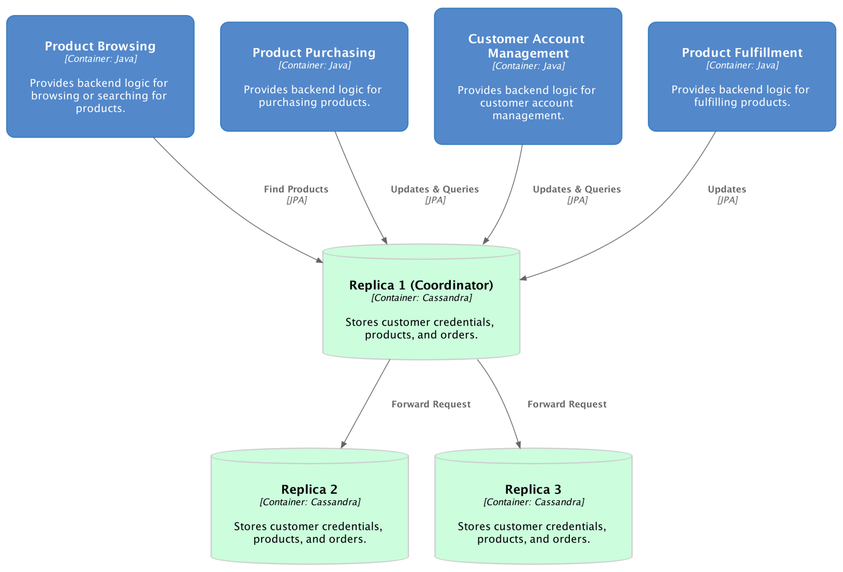

Product Browsing
[Container: Java]
Provides backend logic for browsing or searching for products.
Product Purchasing
[Container: Java]
Provides backend logic for purchasing products.
Customer Account Management
[Container: Java]
Provides backend logic for customer account management.
Product Fulfillment
[Container: Java]
Provides backend logic for fulfilling products.
Find Products [JPA]
Updates & Queries [JPA]
Updates & Queries [JPA]
Updates [JPA]
Replica 1 (Coordinator)
[Container: Cassandra]
Stores customer credentials, products, and orders.
Forward Request
Forward Request
Replica 2
[Container: Cassandra]
Stores customer credentials, products, and orders.
Replica 3
[Container: Cassandra]
Stores customer credentials, products, and orders.

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 (Sharding)

3.1 Partition by Primary Key

3.2 Partition by Secondary Index

3.3 Re-balancing

4 Transactions

4.1 ACID

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.