Log In Join

Back To Module Home

Distributed Systems

0% completed

Search Module

Conclusion

Mark Module as Completed

Multi-Master Replication Algorithm

Look at the multi-master algorithm for replication.

We'll cover the following

As we saw in the previous lesson, single-master replication is a technique that is easy to implement and operate. It can easily support transactions and hide the distributed nature of the underlying system, i.e., when using synchronous replication.

However, single-master replication has some limitations in terms of performance, scalability, and availability.

As we've already discussed, there are many applications where availability and performance are much more important than data consistency or transactional semantics.

A frequently cited example is that of an e-commerce shopping cart, where the most important thing is for customers to be able to access their cart at all times and add items quickly and easily. It is acceptable to compromise consistency to achieve this, as long as there is data reconciliation at some point. For instance, if two replicas diverge because of intermittent failures, the customer can still resolve conflicts during the checkout process.

Multi-master replication#

Multi-master replication is an alternative replication technique that favors higher availability and performance over data consistency.

> This technique is also known as **multi-primary replication**.

In this technique, all replicas are equal and can accept write requests. They are also responsible for propagating the data modifications to the rest of the group.

Multi-master replication has a significant difference from single-master replication. In multi-master replication, there is no single master node that serializes the requests and imposes a single order, as write requests are concurrently handled by all the nodes. This means that nodes might disagree on what is the right order for some requests. We usually refer to this as a **conflict**.

For the system to remain operational, the nodes need to resolve this conflict when it occurs by agreeing on a single order from the available ones.

The following illustration depicts an instance where two write requests can potentially result in a conflict, depending on the latency of the propagation requests between the nodes of the system.

Created with Fabric.js 3.6.6

A client and three replicated nodes A, B, and C

1 of 14

In the case of a conflict, a subsequent read request could receive different results depending on the node that handles the request—unless we resolve the conflict so that all the nodes converge again to a single value.

Conflict resolution#

There are many different ways to resolve conflicts, depending on the guarantees the system wants to provide.

An important aspect of different approaches to resolving conflicts is whether they do it *eagerly* or *lazily*.

- In the **eagerly** case, the conflict is resolved during the write operation.
- In the **lazily** case, the write operation proceeds to maintain multiple, alternative versions of the data record that are eventually resolved to a single version later on, i.e., during a subsequent read operation.

Approaches to conflict resolution#

Here are some common approaches to conflict resolution:

Exposing conflict resolution to the clients#

When there is a conflict, the multiple available versions return to the client. The client then selects the right version and returns it to the system. This resolves the conflict.

An example of this is the shopping cart application, where the customer selects the correct version of their cart.

Last-write-wins conflict resolution#

Each node in the system tags each version with a timestamp, using a local clock. During a conflict, the version with the latest timestamp is selected.

However, this technique can lead to some unexpected behaviors, as there is no global notion of time. For example, write A can override write B, even though B happened “as a result” of A.

Causality tracking algorithms#

The system uses an algorithm that keeps track of causal relationships between different requests. When there is a conflict between two writes (A, B) and one is determined to be the cause of the other one (suppose A is the cause of B), then the resulting write (B) is retained.

However, there can still be writes that are not causally related, i.e., requests are actually concurrent. In such cases, the system cannot make an easy decision.

> We’ll elaborate more on some of these approaches later in the chapters about time and order.

Single-Master Replication Algorithm

Next

Quorums in Distributed Systems

Mark as Completed

Report an Issue