October 14, 2020

Data replication in distributed systems

I've been reading the book Designing Data-Intensive Applications by Martin Kleppmann. Chapter 5 of the book takes a deep dive into the world of data replication in distributed systems. I have always carried the assumption that data replication is non-trivial but reading the chapter opened my eyes to its true complexity. This post is a summary of the chapter - it can be considered a set of notes. Let's do this 🚀

What is data replication?

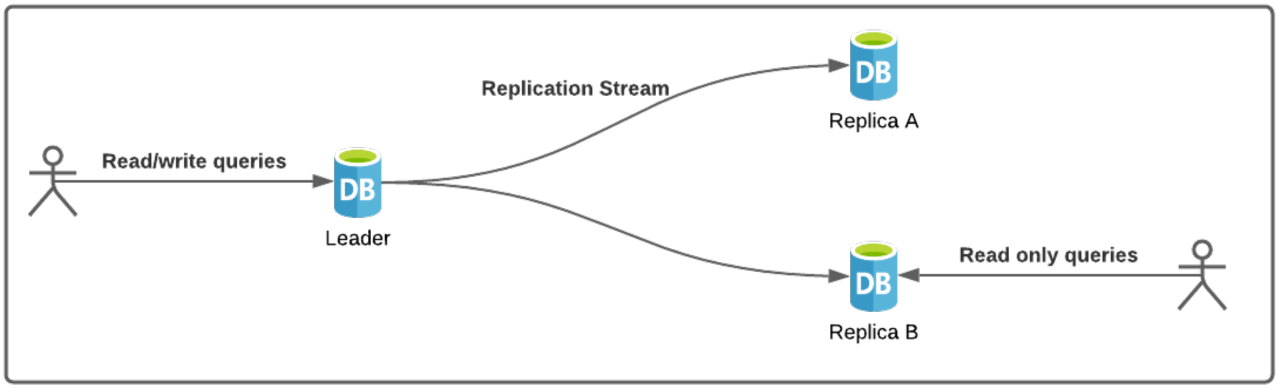
Data replication is the act of copying the *same* data across multiple machines. This is common practice in the development of highly-available software systems. The two leading reasons for data replication:

- In the event one or more databases stop working, the system can continue operating
- To improve the performance of the system. By having replicas, workloads can be distributed across them

The complexity of data replication comes down to managing changes to the data. Propagating changes to all replicas and guaranteeing they have arrived at the same state is non-trivial. The three main algorithms for replicating changes are: single-leader, multi-leader and leaderless replication.

Single-leader replication

The most common solution for replicating changes is *leader-based replication*. In leader-based replication, all writes are submitted to the primary database, a.k.a the leader. The leader sends the changes to the replica databases, a.k.a the followers. This strategy ensures that all the databases contain the same data. The benefit is that reads can be handled by any of the databases (the leader or the followers). This improves the performance of the overall system as the workload is distributed across all the databases.



Synchronous vs asynchronous replication

Data replication is performed in either a synchronous or asynchronous fashion. In synchronous replication, a request is only successful after the last replica has successfully completed the request. This has the advantage of ensuring that data has been successfully replicated across all the replicas. The disadvantage is that if any replica fails to process the request (for example due to a networking issue), the request fails even though some of the replicas may have the relevant data. Alongside this, if any of the replicas are slow to process the request, the response time becomes unacceptable. In asynchronous replication, the request is successful once the leader has processed it. This is advantageous as requests are handled quickly (unless the leader itself is under strain) and writes can still be processed if any number of replicas fail. However, this pattern means writes are not durable; if the leader fails and is not recoverable, writes that have not been replicated are lost.

How to deal with failure

As mentioned, a core advantage of using replication is to improve the availability of the system - it can operate even if replicas become unavailable. There are two scenarios we have to consider: when a *replica* fails and when a *leader* fails.

Replica failure

Every replica keeps a log of the data changes it has received from the leader on disk. If it fails, it uses its log to identify the last transaction it successfully processed. It can then request all subsequent changes it has not processed from the leader and apply them. Once it has recovered, it can continue receiving requests. This process is called *catch-up recovery*

Leader failure

Leader failure is more difficult to handle. If a leader fails, one of the replicas must be promoted to leader, client requests need to be routed to the new leader and the replicas must consume data changes from the new leader. This process is known as *failover*. Failover comes with its own set of challenges:

- If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed. If the old leader rejoins, the extra writes it has processed have to be dealt with. Commonly this is done by discarding the writes. Unfortunately, this negatively impacts the durability guarantee.
- It's possible for two replicas to believe they are the leader. This is known as *split-brain*. If both begin accepting writes and there is no method for resolving write conflicts, data can be lost or corrupted.
- How do we know for certain that a leader is dead? Often a timeout is used; if the leader does not respond in a certain amount of time, it is declared dead. If this timeout is too short, we run the risk of performing unnecessary failovers. If it is too long, the system takes longer to recover.

Under the hood of leader-based replication

By now, I'm sure you're curious to know how replication is implemented. There are a few common methods.

Statement-based replication

In statement-based replication, the SQL statement itself is passed onto replicas. When a write request is received by the leader, it logs the SQL statement and propagates it to the replicas. This approach has a few failure modes:

- If a statement contains a nondeterministic function, the writes will be different. For example, if the statement needs the current time, each replica will have a slightly different time.
- Statements that have side effects (e.g a webhook) will be repeatedly called. This is not ideal as it is a waste of computation and may lead to different results if the side effect is nondeterministic.

Write-ahead-log shipping

Databases usually keep a log containing information of every write processed. This log can be used to build a replica of the database. Using this method, the leader writes every write to the log and propagates the log to each replica. The replica processes the log and builds the data as it is on the leader. The disadvantage with this method is that a log usually describes the data on a low-level - it contains details of which bytes were changed in which disc blocks. This tightly couples the replication process and the storage engine; if the database storage format changes, you have to upgrade the entire fleet of databases at once since the leader and followers cannot have different versions. This makes it impossible to have zero-downtime upgrades - this is unacceptable for many companies.

Logical log replication

To decouple the replication process and storage engine, different log formats can be used for the storage engine and replication. The replication log is called a *logical log*. This log contains information about writes to the database at a row level.

- For an inserted row, the log contains new values for all columns
- For a deleted row, the log contains enough information to identify the

deleted row

- For an updated row, the log contains enough information to identify the updated row and the new values for the changed columns

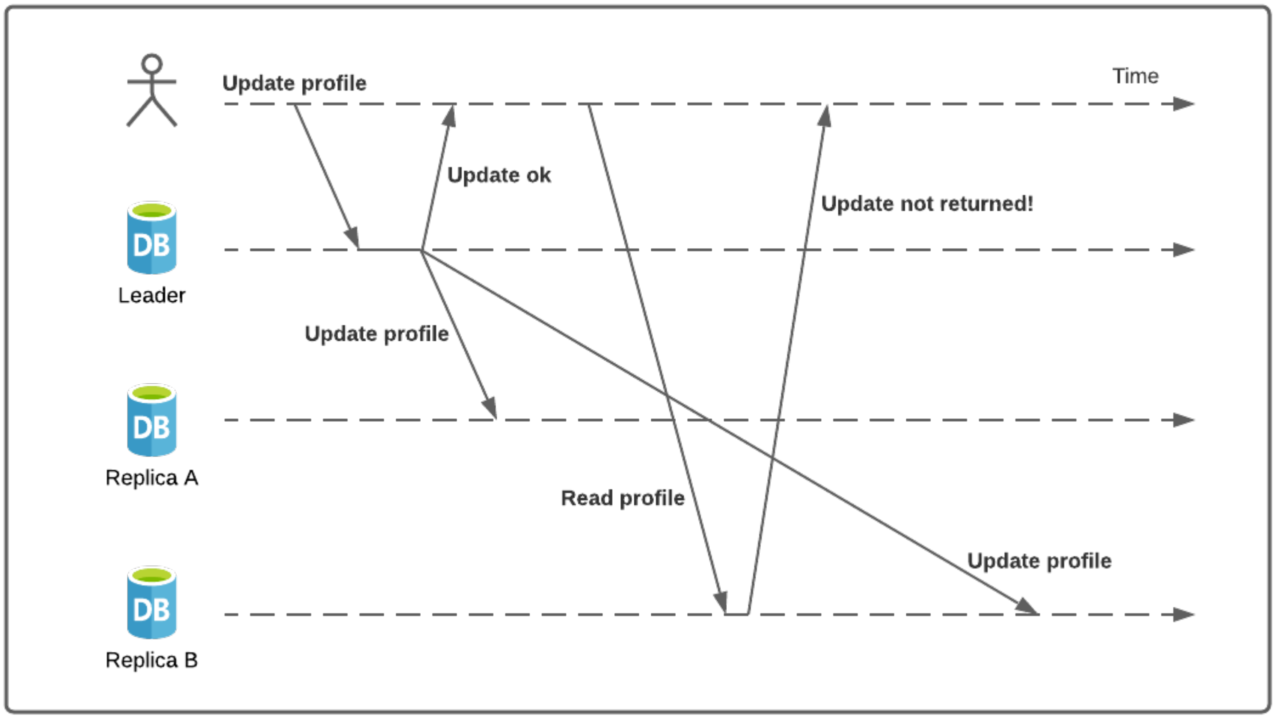
Using a logical log allows for zero-downtime upgrades. Another benefit is that it is easier for external applications to parse a logical log. This proves useful if there is a need to send the contents of a database to an external system such as a data warehouse.

Replication lag

Leader-based replication is great for workloads that mainly consist of reads and few writes. The leader is responsible for processing all the writes while read requests are distributed amongst replicas. This is known as a *read-scaling* architecture. It is only practical with the use of asynchronous replication. When a write is made, there is a replication lag - the time it takes for the write to be propagated and applied by the replica. After some time (usually a fraction of a second), all replicas will have the same data and *become consistent*. This is known as *eventual consistency*. Eventual consistency brings its own set of challenges.

Read your own writes

Imagine a scenario where a user updates her profile and views it immediately after. The update is a write request which is handled by the leader and propagated to the replicas. The viewing of her profile is a read request and can be handled by any of the replicas. It can so happen that the replica has not processed the write by the time she views her profile, showing her stale data.

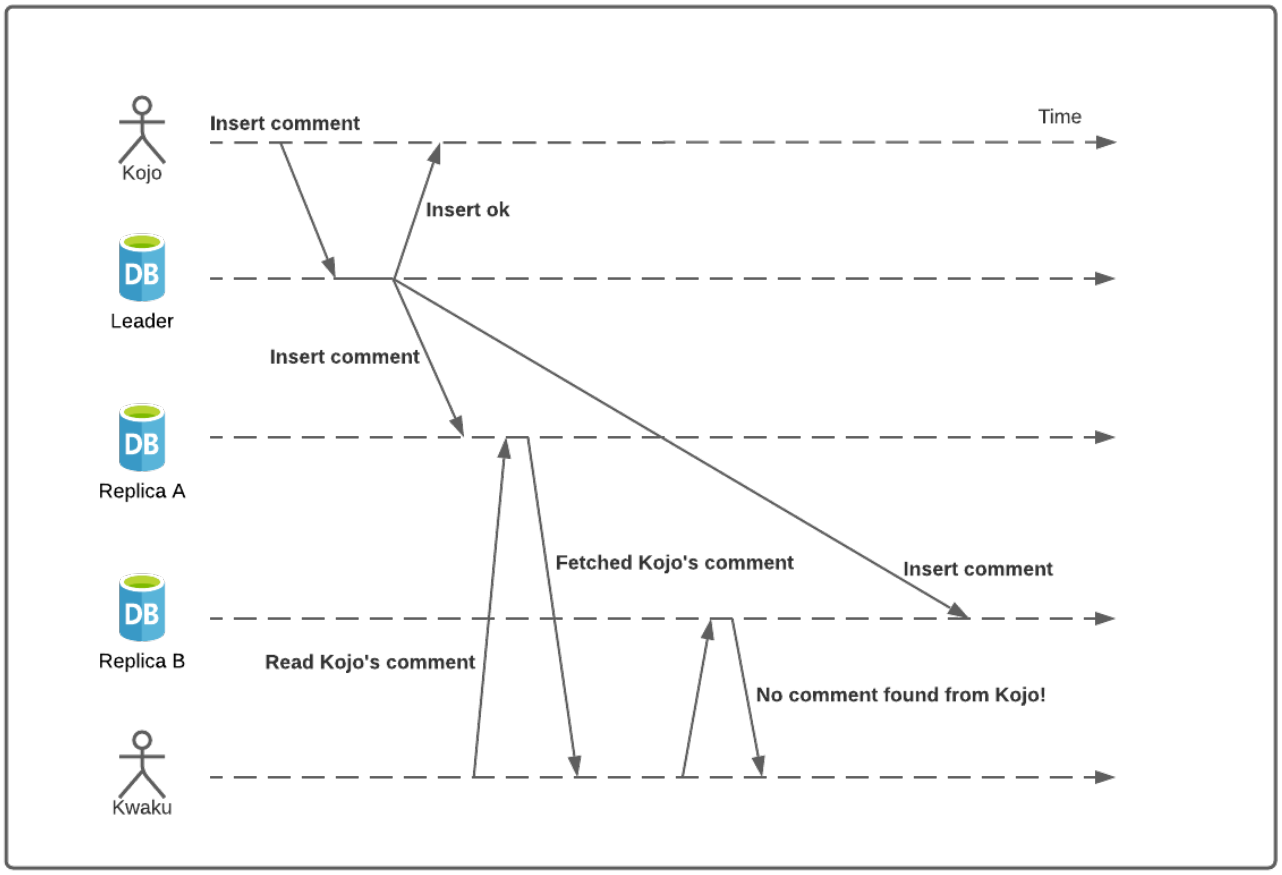


This situation requires *read-after-write consistency*, also known as *read-your-writes consistency*. This ensures that a user will always see the updated data they submitted *themselves* but makes no claim as to what state the data will be for other users i.e the data you view of other users may be stale. There are several ways to implement this:

- If you are reading something the user modified, read it from the leader otherwise read it from the follower. This requires you to know what data was modified. There are simple heuristics one can employ instead of tracking this. For example, a user is the only person that can edit their own profile so a simple rule is: always read a user's own profile from the leader
- Keep track of when an update was made. For a window period (e.g one minute), read from the leader. After that period has elapsed, read from the follower.
- The client keeps track of its latest write via a timestamp. The system can ensure whenever a read request occurs, the replica serving the request has data up until that timestamp. If it does not, it can be serviced by another replica or the leader.

Monotonic reads

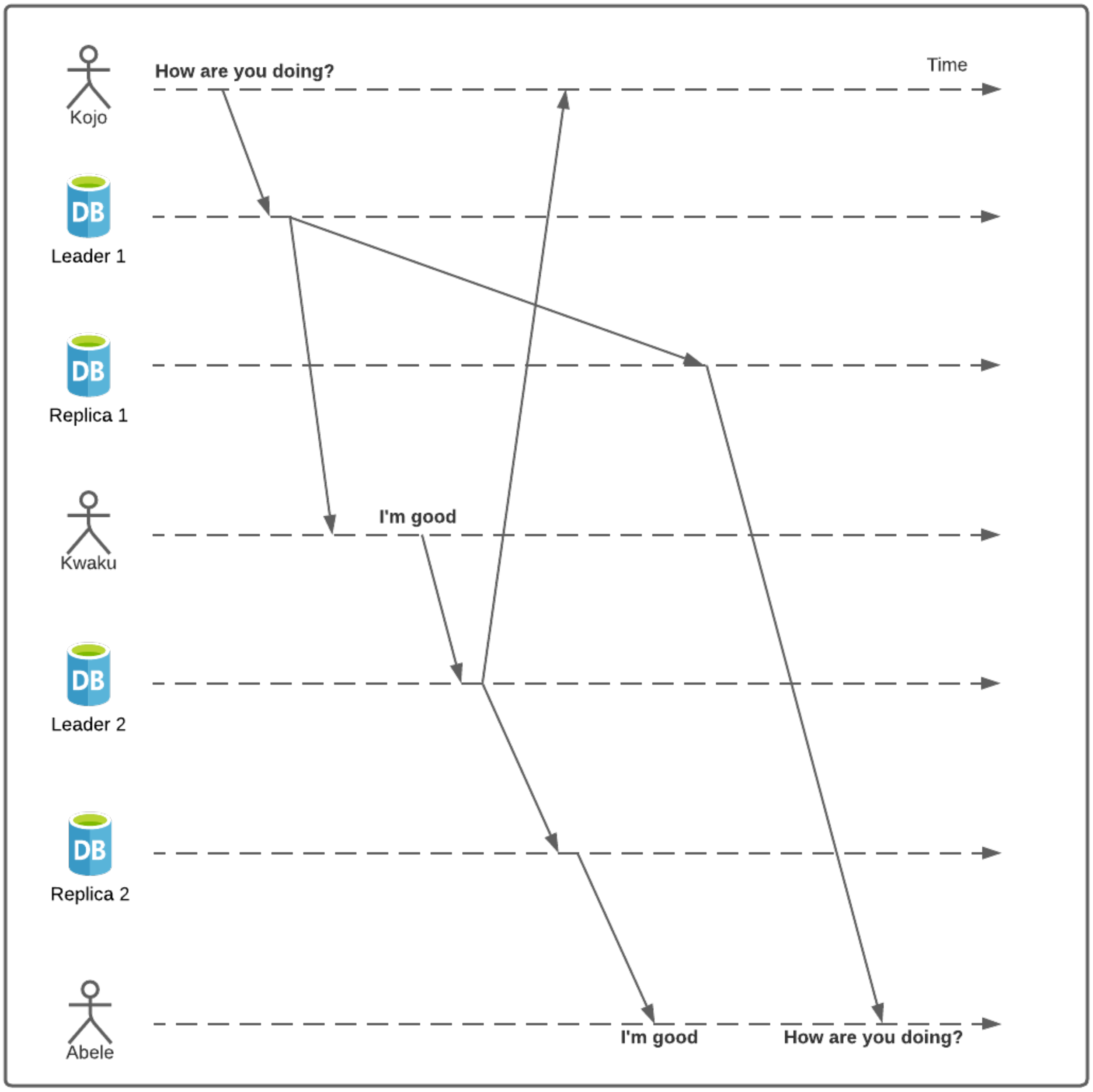
If a user makes several reads from different replicas, it's possible for a user to see things go back in time. Time travel, how rad 😆. Let's take an example scenario. A user, Kojo, makes a new comment. This is processed by the leader and is propagated to replicas A and B. Replica A applies the write immediately but there is a lag with replica B. Another user, Kwaku, reads the post where the comment was made. This read request is handled by replica A and so Kwaku sees the new comment. He then refreshes the post. This time, the read is handled by replica B but this replica is suffering from a lag and therefore hasn't processed the comment yet. When Kwaku refreshes the post, the comment is gone! In this sense, he's time-traveled to the past 🚀.



Monotonic reads are a guarantee that this effect will never happen i.e if they make multiple requests, they never see time go backward. One method for achieving this is to simply route requests to the same replica for a given user.

Consistent prefix reads

A class of errors can be attributed to violating causality. In this scenario, Kojo and Kwaku are chatting. Another user, Abele is observing the conversation. Kojo asks Kwaku, “How are you doing?”. Kwaku replies to Kojo saying, “I’m good”. So we have two writes that need to occur. In our database design, there is a leader and follower for Kojo and a separate leader and follower for Kwaku. Since Abele is just reading, his reads come from the respective followers. When Kojo sends his message, it is processed by the leader but the replication lag is longer than normal. Kwaku replies and his message is processed by his leader and is immediately processed by the replica. When Abele reads, he gets Kwaku’s reply first and then later gets Kojo’s initial question. The causal link between the messages is now broken.



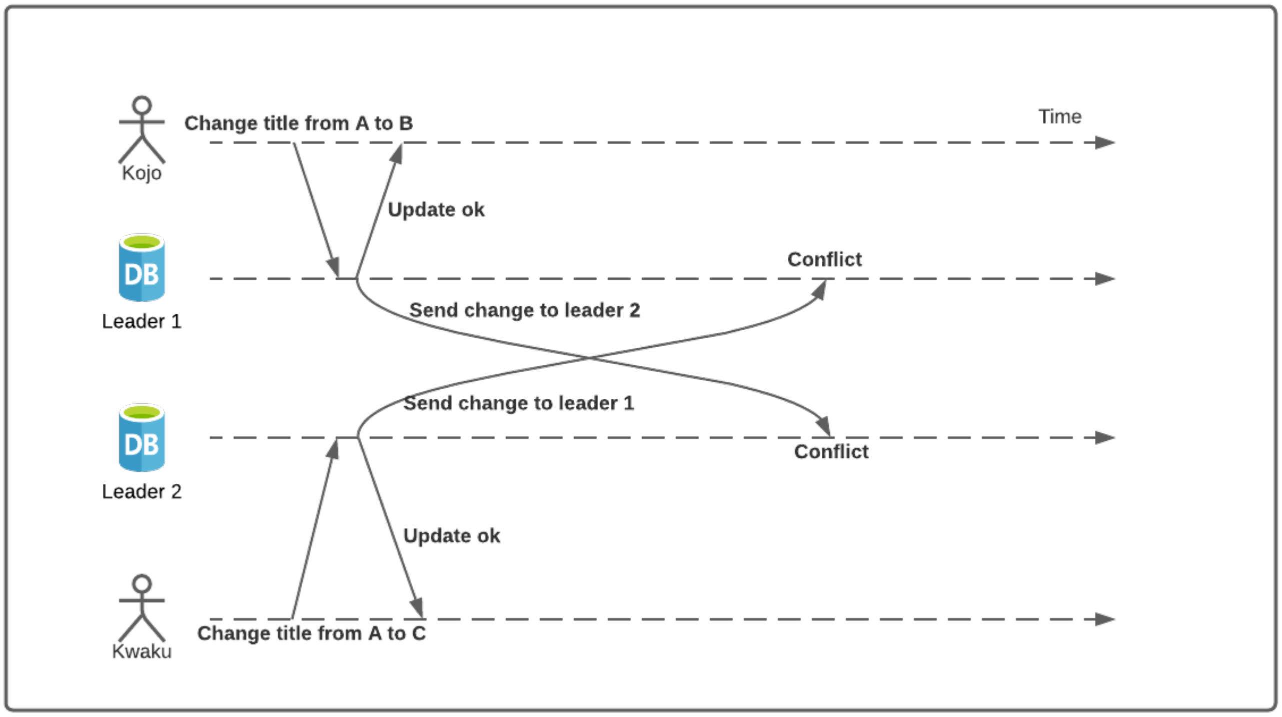
Consistent prefix reads are a guarantee that anyone reading a set of writes reads them in the order they were written. This problem is magnified in partitioned databases. Partitions act independently; there is no global ordering of writes to the system. When a user reads, they may see parts of the database in an older state and others in a newer state. A solution to this is to write all causally related writes to the same partition.

Multi-leader replication

One obvious shortcoming with single-leader replication is that there is only a single leader. All writes have to be processed by this leader. If it goes down or cannot be reached, you cannot process any writes. A simple extension is to allow for multiple replicas to be leaders. In this setup, a write can be processed by any of the leaders and the changes are propagated to all other replicas (including the other leaders). A multi-leader setup introduces additional complexity. This often outweighs the advantages of using a multi-leader setup, however, there are several use cases where it reasonable to do so. An example is a multi-datacenter database. In this setup, a database is replicated across different datacenters. This is often to handle an entire datacenter going offline. In a single-leader setup, the leader will be in only *one* datacenter and the changes will need to be replicated across all the datacenters from there. In a multi-leader setup, *each* datacenter has its own leader - the single-leader configuration exists within each datacenter. When a write occurs in a given datacenter, the leader propagates the changes to the other leaders in the other datacenters.

Write conflicts

One of the biggest challenges with a multi-leader setup is handling write conflicts. Let’s take a scenario where Kojo and Kwaku are collaboratively editing a document. Kojo changes the title of the document from A to B. Kwaku also changes the title from A to C at the same time. Both changes are accepted by their local leaders but when each leader propagates the change to the other database, a conflict occurs.



The simplest method of handling write conflicts is avoiding them. If an application ensures that all writes go through the same leader, a write conflict cannot occur. Surprisingly, this is often the recommended approach as dealing with write conflicts is complex. Some other strategies for handling conflicts:

- Give each **write** a unique ID (e.g timestamp). The one with the highest number ID is used and the other writes are thrown away. This is known *last-write-wins*.
- Give each **replica** a unique ID. The one with the highest ID takes precedence over the others. Its write is applied while the others are discarded.
- Record the conflict elsewhere and write application code that handles the conflict.

Leaderless replication

In leaderless replication systems, any replica can process writes. The advantage of a leaderless system is improved fault-tolerance.

Writing when a node is down

In a leaderless system, writes are sent to each replica in parallel. Imagine a scenario where we have 3 replicas and 1 is down. When a write request enters the system, two of the three replicas successfully process the request. The system is configured such that if two out of three replicas process the request, the request is considered successful. When the unavailable replica recovers and comes online again, if it receives read requests it will return stale data. To solve this, read requests are sent in parallel to each replica and the most recent value is returned to the client.

Achieving consistency

It is important to achieve eventual consistency - all replicas have the most recent data. There are two common mechanisms for this, *read repair* and *anti-entropy process*. In read repair, the database system can detect stale data on read and subsequently update it. This is an effective technique when data is read frequently. An anti-entropy process is a continuous process that looks for differences in data across all replicas and copies any missing data to stale replicas to keep them up to date.

Quorum

In the scenario above, we considered a write request successful if two out of three replicas processed the request. This may seem counter-intuitive; how do we guarantee we will never receive stale data? If we know that every successful write is guaranteed to be present on at least two out of three replicas, that means at most one replica can be stale. If a read request is processed by at least two replicas we can guarantee one of the responses will be up to date.

We can formalize this concept. If there are n replicas, each write must be processed successfully by w nodes and reads must be processed by at least r nodes. We can guarantee up to date reads if $w + r > n$. For our example, we had $n = 3, w = 2, r = 2$. This satisfies the constraint. Reads and writes that abide by this constraint are called *quorum* reads and writes. The quorum condition allows the system to process requests if nodes are unavailable:

- If $w < n$, we can still process write requests

- If $r < n$, we can still process read requests

If fewer than r or w replicas are available, then we cannot process reads or writes.

Limitations of quorum consistency

Even with the quorum condition satisfied, there are edge cases that can arise. These are often unrelated to a quorum itself but the nature of distributed systems.

- If a sloppy quorum is used, the replicas written to and read from are not guaranteed to overlap.
- If a write and read happen concurrently, it is undetermined whether the read response is of the old or new value.
- If a write succeeds on fewer than *w* replicas, the write is not rolled back on the replicas it succeeded on. When a read request occurs, it may or may not return the value from the read
- If a replica with a new value fails and is restored from a replica with an old value, the number of replicas storing the new value may fall below w.

Sloppy Quorum

As mentioned earlier, quorums are not as fault-tolerant as they are said to be due to the nature of distributed systems. This results in some difficult design decisions. Imagine we have a cluster with 9 replicas. Even though we have 9 replicas, we set n to 3 and $w = r = 2$. In this scheme, we effectively have 3 partitions each containing 3 replicas. Of that, a particular value only needs to exist on one of those partitions and satisfy the quorum condition to ensure we always have up to date responses. This, however, can go wrong. A network interruption occurs which cuts off a client to some replicas. The replicas its cut off from contain *some* of the replicas needed to guarantee the quorum condition is met. For example, of the 9 replicas, it is cut off from 4 and 2 of those are from one partition needed to ensure a quorum for a particular read request. In this case we have a design choice:

1. We return errors to all requests as we cannot reach a quorum
2. We process write requests and write to them to nodes that are reachable

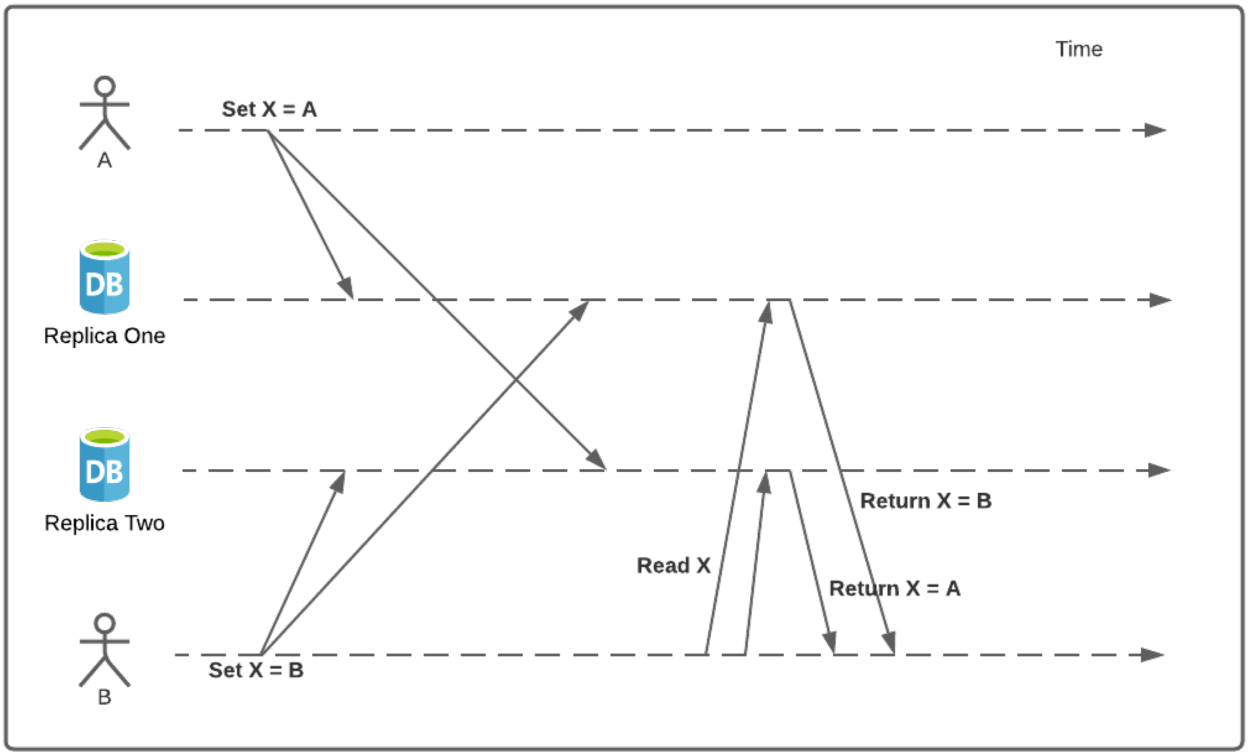
but are not part of the partition where the value is located

The second option is referred to as a *sloppy quorum*. We still need to satisfy the quorum condition but there is no guarantee the replicas are from the partition where the value is located. Once the other replicas are online again, the writes are sent to those replicas. This is known as *hinted handoff*. Sloppy quorums are useful when there is a requirement for high-availability for writes. In reality, a sloppy quorum is not a quorum - it does not guarantee you will read the most up to date value. It is a guarantee of durability - the data will be stored on w nodes somewhere.

Detecting concurrent writes

Many leaderless replication systems allow for concurrent writes. This will inevitably lead to write conflicts. The main problem is that the order of writes may differ at different replicas. This can be due to a multitude of factors: variable network delays, network interruptions, partial failures, etc. In the image below, we have two replicas and two clients, A and B.

- Replica one receives the request from A and then B.
- Replica two receives the request from B and then A.



The concurrent writes have lead to differing end results in each replica. When a read occurs, replica one will return B but replica two will return A. In order to be eventually consistent, the nodes should converge to the same value. There are some methods for handling this.

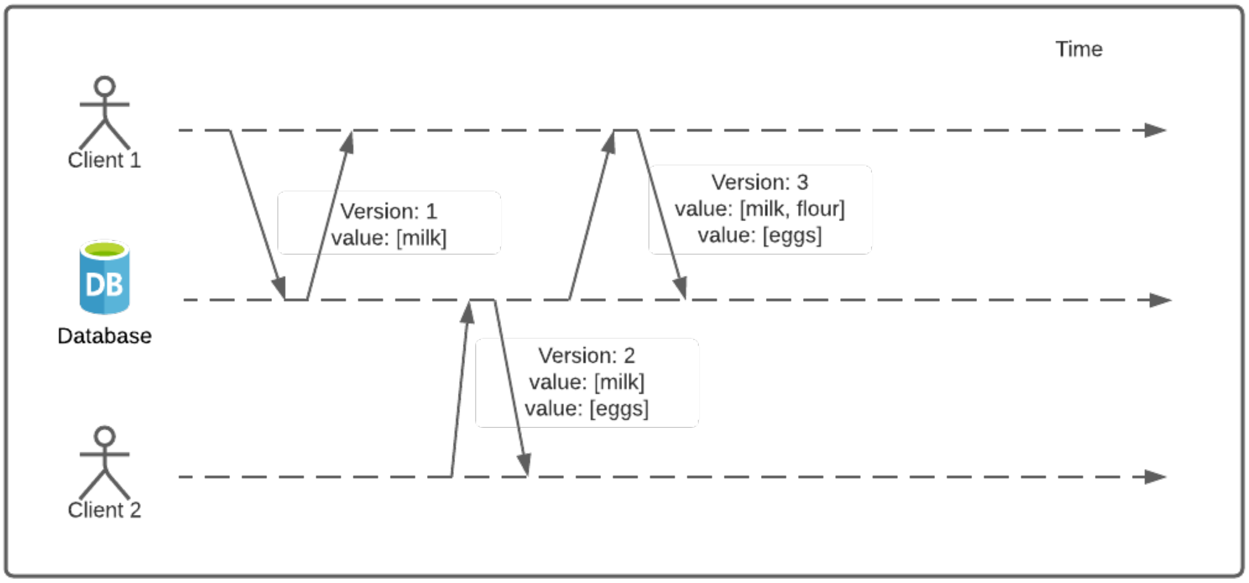
Last write wins

Last write wins uses the most recent write as the convergent value and discards the other values. While it can be implemented, it does not make much sense. As the writes happen concurrently, there is no true "most recent" value. It is done by enforcing an order on the writes. For example, we can use timestamps to enforce this order. Unfortunately, last write wins trades-off durability - data is lost in the conflict resolution process.

Version Vectors

Version vectors use a set of version numbers to deal with concurrent writes. A version number is a monotonically increasing number that is incremented every time a write operation occurs. This version number is sent with all write requests, allowing the system to determine if a concurrent write occurred and then apply a strategy to deal with it. Taking the example from the book, we have two clients buying groceries on an online store, concurrently adding items to the same shopping cart.

1. Client 1 adds *milk* to the cart. This is the first write so the version number is incremented to 1 and the cart is updated to **[milk]**. The version number and the cart state are sent back to the client.
2. Client 2 adds *eggs* to the cart concurrently. The client did not know milk had been added to the basket already. The version number is incremented to 2 and the milk and eggs are stored as two *separate* values. Therefore the value of the cart is **[[milk], [eggs]]** The server sends back the version number and both the values of the cart.
3. Client 1 wants to add *flour* to the cart. From their perspective, the cart should contain **[milk, flour]** after the update. When requesting to add the flour, the version number and request are sent to the server. The version number is the one stored on the client-side - in this case it is 1. The database can determine that a concurrent write occurred. The value **[milk, flour]** occurs after the value of **[milk]**. However, considering the other client added eggs to the cart and that is not contained in the new value, it is straightforward to see that the request to add flour occurred concurrently with adding eggs to the cart. The value **[milk, flour]** is assigned version number 3. Version number 1 and its value are overwritten. Version number 2 and its value **[eggs]** is kept.



Version vectors are an extension of this, they contain version numbers of each replica. This ensures it is safe to read from one replica and subsequently write back to another replica.